WINDCREST®/McGRAW-HILL

VISUAL BASIC™ FOR DOS

- Disk includes project and form files
- Learn to document forms, modules, controls

Namir C. Shammas

Visual Basic™ for DOS

Visual Basic™ for DOS

Namir C. Shammas

Windcrest®/McGraw-Hill

Dedication

To my grandmother Emily,
while we've never met,
I feel I know you,
through the thorn we both share.

FIRST EDITION
FIRST PRINTING

© 1993 by **Windcrest Books**, an imprint of TAB Books.
TAB Books is a division of McGraw-Hill, Inc.
The name "Windcrest" is a registered trademark of TAB Books.

Printed in the United States of America. All rights reserved. The publisher takes no responsibility for the use of any of the materials or methods described in this book, nor for the products thereof.

Library of Congress Cataloging-in-Publication Data

Shammas, Namir Clement, 1954-
Visual Basic for DOS / by Namir C. Shammas.
p. cm.
Includes index.
ISBN 0-8306-4375-3 (pbk.)
1. BASIC (Computer program language) 2. Microsoft Visual BASIC.
3. MS-DOS (Computer file) I. Title.
QA76.73.B3S448 1992
005.26′2—dc20 92-32303
CIP

Acquisitions Editor: Brad Schepp
Editor: Melanie Brewer
Director of Production: Katherine G. Brown
Book Design: Jaclyn J. Boone
Cover Design: Sandra Blair Design
and Brent Blair Photography, Harrisburg, Pa. WP1

Contents

Acknowledgments *xiii*

Introduction *xv*

1 In the beginning *1*

DOS and OOP 1
Object-oriented programming basics 2
The Visual Basic features 3
The visual controls 3
The Visual Basic control panel 4
Event-driven programming in Visual Basic 6
The properties of an object 8
Starting Visual Basic 9

2 The Visual Basic Help system *11*

About hypertext 11
Navigating through the Help system 12
Context-sensitive help 14
Additional help 14

3 Your first applications *15*

The first Visual Basic application 16
Visual Basic projects 16
Drawing controls in the form 17
Altering the control properties 18
Attaching event-handling procedures 19

Saving your project 21
Opening your project 22
The second application 22
The third application 25

4 *Drawing the application form* 29
The ToolBox 30
The command-oriented scientific calculator 32
Drawing the controls 34
Drawing a control 35
Resizing a control 36
Moving a single control 37
Moving multiple controls 37
Deleting a single control 37
Deleting multiple controls 38
Aligning the controls 38
Saving your work 38
Documenting the Visual Basic forms 38

5 *Changing control values* 43
Working with the Properties bar 43
Viewing properties and values 43
Changing the current value 47
Documenting custom values 48
Changing the values for the scientific calculator 49

6 *Attaching code to the controls* 53
Coding a Visual Basic application 53
Writing event-handling procedures 54
Working with the code window 56
Inserting code in CONSTANT.BI file 57
Attaching code to the controls 57
Running the COSC application 64
Creating an EXE file 64

7 *Text boxes and Command buttons* 67
Text boxes 67
Command buttons 69
A text file viewer 70
The program's functionality 70
The program code 74
Running the file viewer 78
Setting access keys to command buttons 78
Manipulating the tab order of controls 79
Emulating scrollable DOS screens 80

The basic approach 80
The code for the first pseudoscreen 81
The PSCREEN project 86

8 Application menus 91

Definitions 91
The Menu Design Window 91
Building a bare-bones menu 93
Controlling menus at run time 95
The menu-drive text file viewer 96
Assigning shortcut and access keys to menus 98
Control arrays 102
Control arrays in menus 104
Control array menus for the file viewer 105

9 Working with multiple forms 111

Dialog boxes and modal forms 111
Methods and statements that manage forms 112
Setting the form properties 113
Selecting the startup form or module 113
An example 114

10 Form input 125

Monitoring text box input 125
Option buttons 132
Check boxes 137
List and combo boxes 140
Vertical and horizontal scroll bars 144
The INPUTBOX$ and MSGBOX dialog boxes 150
Updating the scientific calculator 153

11 Output 163

Output to a form, picture box, and the printer 163
The Visual Basic PRINT statement 164
The FORMAT$ function 165
Printer output 165
A Pseudoscreen in a picture box 165
The Calendar form 168
The Text File printer 176

12 Managing mouse events 181

The mouse events 181
The free-drawing program 182
The mouse-pointer demo program 185
Dragging and dropping controls 190

13 Interactive Visual Basic programs 197

Using the CLIPBOARD 197
Timer controls 207
Developing low-level keyboard handlers 211

14 Run time errors 215

Defensive programming techniques 215
Visual Basic error handling 216
The types of run time errors 218

15 Working with the file system controls 223

The drive list box 223
The directory list box 224
The file list box 225
The directory tree viewer 226
The text editor 232

16 File I/O 241

Common file I/O 241
Sequential file I/O 243
Listing sort utility 245
Random-access file I/O 258
The contacts database application 259
Binary file I/O 266
The text sorting utility 275

17 Providing on-line help 283

The help compiler 283
Offering on-line help 292

A Visual Basic for DOS language: Quick reference 305

Data types 305
Literal constants 305
Operators 306
Identifiers 306
Symbolic constants 308
Variables 308
Arrays of variables 309
    Two-dimensional arrays 311
    Multidimensional arrays 311
    Dynamic arrays 312
Control arrays 312
User-defined data types 313
Decision-making control structures 314

The IF-THEN structure 315
The IF-THEN-ELSE structure 315
The SELECT-CASE structure 317
Loop control structures 319
The FOR-NEXT loop 319
The DO-WHILE-Loop 321
The DO-UNTIL-Loop 322
The DO-LOOP 323
Exiting from loops 324
Nesting control structures 324
SUB and FUNCTION procedures 325
Sub procedures 325
FUNCTION procedures 328
Exiting procedures 329
Static Local Variables 330
Form arguments 331
Control arguments 332
Modules 333

B The companion disk *337*

Index *339*

Acknowledgments

This book is the fruit of the support and encouragement from a lot of people. I would like to thank Brad Schepp of TAB Books for his backing, encouragement, and shared vision for the project. I am also very grateful to Erin Carney and Troy Strain at Microsoft for their support. I would also like to thank the executive editor Bob Ostrander and book editor Melanie Brewer for their efforts in shaping up the book.

Introduction

The Visual Basic for Windows has ushered a new era of programming for the masses, namely, visual programming. The huge success of Visual Basic for Windows has prompted Microsoft to bring out a DOS version, Visual Basic for MS-DOS. Using Visual Basic for DOS you can develop professional looking DOS applications quickly and with great ease. This book shows how you can learn to program effectively with Visual Basic for DOS. The book focuses on the visual interface of programs written using Visual Basic. The ISAM, graphics, and presentation graphics that are part of the Visual basic for DOS package are not covered.

The book contains 17 chapters that present various topics of programming in Visual Basic. The chapters can be divided into three logical groups: introductory, intermediate, and advanced.

Chapter 1 offers a technical introduction to the basic notions and concepts behind programming Visual Basic.

Chapter 2 explains how to use the built-in help system. This system uses hypertext technology and allows you to obtain a good amount of information without resorting to the Visual Basic manuals.

Chapter 3 gives you a summary of the three basic stages involved in building a Visual Basic application. The chapter offers three programs that give you a basic idea on how to program Visual Basic applications.

Chapters 4, 5, and 6 discuss in depth the steps involved in writing a Visual Basic application. These chapters also introduce you to special program documentation that I use throughout the book.

Chapter 7 focuses on the text box object. This is a versatile control that allows you to input and view text. The chapter also discusses how to use the text box control to emulate scrollable screens.

Chapter 8 discusses creating and using Visual Basic menus. Visual Basic offers an easy way to design menus, including nested menu options.

Chapter 9 deals with multiform applications. Using multiple forms allows you extend the interface of your applications by offering special dialog boxes and visual forms.

Chapter 10 discusses the various ways to obtain input from the user. This includes using dialog boxes, check boxes, option buttons, and scrollable bars.

Chapter 11 presents the various ways that a Visual Basic program can provide unformatted and formatted output. The chapter also discusses how you can use a visual form to emulate a screen.

Chapter 12 discusses using the mouse more effectively in Visual Basic application. Two programs illustrate the various mouse manipulation features offered by Visual Basic.

Chapter 13 looks at writing Visual Basic programs that are more interactive. This includes using the clipboard object, using timer controls, and detecting any key on your keyboard, including function keys and cursor control keys.

Chapter 14 focuses on error handling in Visual Basic programs. The chapter presents the built-in error-handling mechanism and discusses the various type of error-handling schemes.

Chapter 15 presents the visual controls that make up the Visual Basic file system. These controls allow you to select the current drive, current directory, and a file. The chapter offers a simple text editor to demonstrate the controls of the file system.

Chapter 16 discusses using files for input and output. This includes sequential files, random-access files, and binary files. The chapter presents program examples that can serve as utilities.

Chapter 17 deals with offering context sensitive online help in your Visual Basic applications. The chapter presents an example help form that you can easily use in your own applications.

Appendix A provides a quick reference for the Visual Basic language. Appendix B briefly discusses installing the accompanying disk.

This book shows you how to write nontrivial programs. For example, the book contains programs that are text file viewers and text file editors. Such programs enable you to see how the various visual controls in Visual Basic work together.

Icons are used throughout the book to help you identify certain aspects of Visual Basic faster. You can recognize these icons right away:

These icons show where to find warnings.

These icons point out things you should know.

These icons precede programming notes or tips.

1
In the beginning

Microsoft Windows is a user interface that is more sophisticated and easier to use than DOS. At the same time, the Windows interface has become an inspiration for the next generation of DOS utilities. These utilities offer a better user-interface that supports the mouse, a desktop, and windows with controls. All these are text-based.

Visual Basic for DOS is the DOS version of Visual Basic for Windows. Throughout this book I will refer to the Visual Basic for DOS as simply Visual Basic. Visual Basic also includes elements from the popular QuickBasic and Microsoft Professional Basic compilers.

You should be acquainted with programming in QuickBasic to fully understand this book. (Appendix A contains a quick language reference for Visual Basic.) Visual Basic empowers you to develop professional looking DOS applications easier and quicker than other languages. The basic notions and components of programming Visual Basic are discussed in this chapter.

DOS and OOP

The computer and software industry goes through cycles of promoting new technology. This is usually accompanied by some media hype and the advent of a new set of buzzwords, which is just what we all need! The year 1986 witnessed the boom of artificial intelligence (AI). This was short-lived, because the hype over AI promised much more than what the state of the art could actually deliver.

A few years ago, the software industry promoted object-oriented programming (OOP)—again with a new set of buzzwords. Unlike AI, object-

oriented programming had a more defined set of objectives and did not promise the moon. Object-oriented programming was billed as superior to structured programming in reusing code and providing more sound program development. This claim was even more substantiated with the advent of Microsoft Windows.

Using a non-OOP popular language like C did not make programming Windows applications any easier. It took multiple pages of C code just to show a simple window with a "Hello World!" message. The advocates of OOP, such as the Whitewater Group that sells the OOP Actor language, kept reminding Windows programmers (and those who aspired to be so) that OOP languages can give the almighty C a run for its money; they were not exaggerating. Software developers started paying more attention to a new successor of C, C++, which offered object-oriented programming features.

The year 1989 also witnessed the advent of OOP language extensions to DOS-based Pascal and Modula-2 compilers. It was only a matter of time before various languages (such as C++, Pascal, and Modula-2), armed with OOP features, were rolled out for programming Windows applications. Not to leave Basic out of the family of Windows-oriented languages, Microsoft offered Visual Basic for Windows. While Visual Basic for Windows lacks formal OOP language extensions, it does implement a number of essential OOP elements.

This approach, along with the implemented visual programming methods, proved to be extremely successful and was very well received by the Windows programmers. This prompted Microsoft to offer Visual Basic to the DOS programmers to enable them to enjoy visual programming methods.

Object-oriented programming basics

Object-oriented programming views the world around as one of objects. Each object has a number of characteristics or attributes and a specified set of functionalities. For example, my CASIO Data Bank watch is an object. It is made up of an LCD display, a miniature keyboard, control buttons, a casing, and the internal electronics. The functionality of the watch includes displaying the date/time, phone numbers, short memos, schedules, a calculator, and an alarm.

The concept of objects can be easily applied to the various visible components of any DOS application (including Visual Basic). For example, each command button in a dialog box is an object, with its display characteristics and underlying functionality.

The example of my CASIO Data Bank watch also serves to illustrate another important programming concept used by Visual Basic, namely, event-driven programming. Let me explain event-driven programming using my CASIO watch. Normally, the watch performs its main function, displaying the current date and time; however, I can press a side button to search for a phone number. Pressing the button sends an electrical signal to the internal logic circuits. The signal is handled by a hardcoded procedure that displays

the main menu for selecting phones, memos, and schedules. Thus, event-driven programming involves generating events and handling them.

This differs significantly from the linear programming method used in QuickBasic. Visual Basic implements event handling through a family of procedures, which you write. These procedures are on standby, ready to deal with a specified set of events. Combining the concepts of objects and event handling makes it even easier to design Visual Basic programs—the visual objects handle the events generated, for example, by clicking on a mouse button or pressing a key.

The Visual Basic features

Visual Basic offers a text-based DOS interface. With Visual Basic you can create single and multiple windows, each with visual control objects. These objects include command buttons, option buttons, check boxes, list boxes, combo boxes, text boxes, scroll bars, frames, menu bars, file selection boxes, directory selection boxes, and drive selection boxes. This impressive list of control objects clearly indicates that Visual Basic offers a quantum leap for QuickBasic programmers. In fact, Visual Basic offers the following features:

- Run-time response to the mouse of the keyboard.
- Showing and hiding control objects during program execution.
- The ability to access a clipboard buffer and the printer.
- Making calls to the DOS functions and assembly language routines.
- The ability to debug your programs during the development phase.
- The ability to test part of your code in an immediate-mode window.

The visual controls

In the last section I mentioned the visual controls that are used in Visual Basic. If you are familiar with these controls then you can skip this section. Otherwise, here is a brief description of these controls:

Label Static text that appears in a window.

Text box A control that accepts user input.

Command button A control that you click to perform a task.

Frame A control that contains other controls, especially check boxes and radio buttons.

Check box A control that you can check or uncheck. The state of a check box is independent of other check boxes.

Radio button A control that you can select and deselect. When multiple ra-

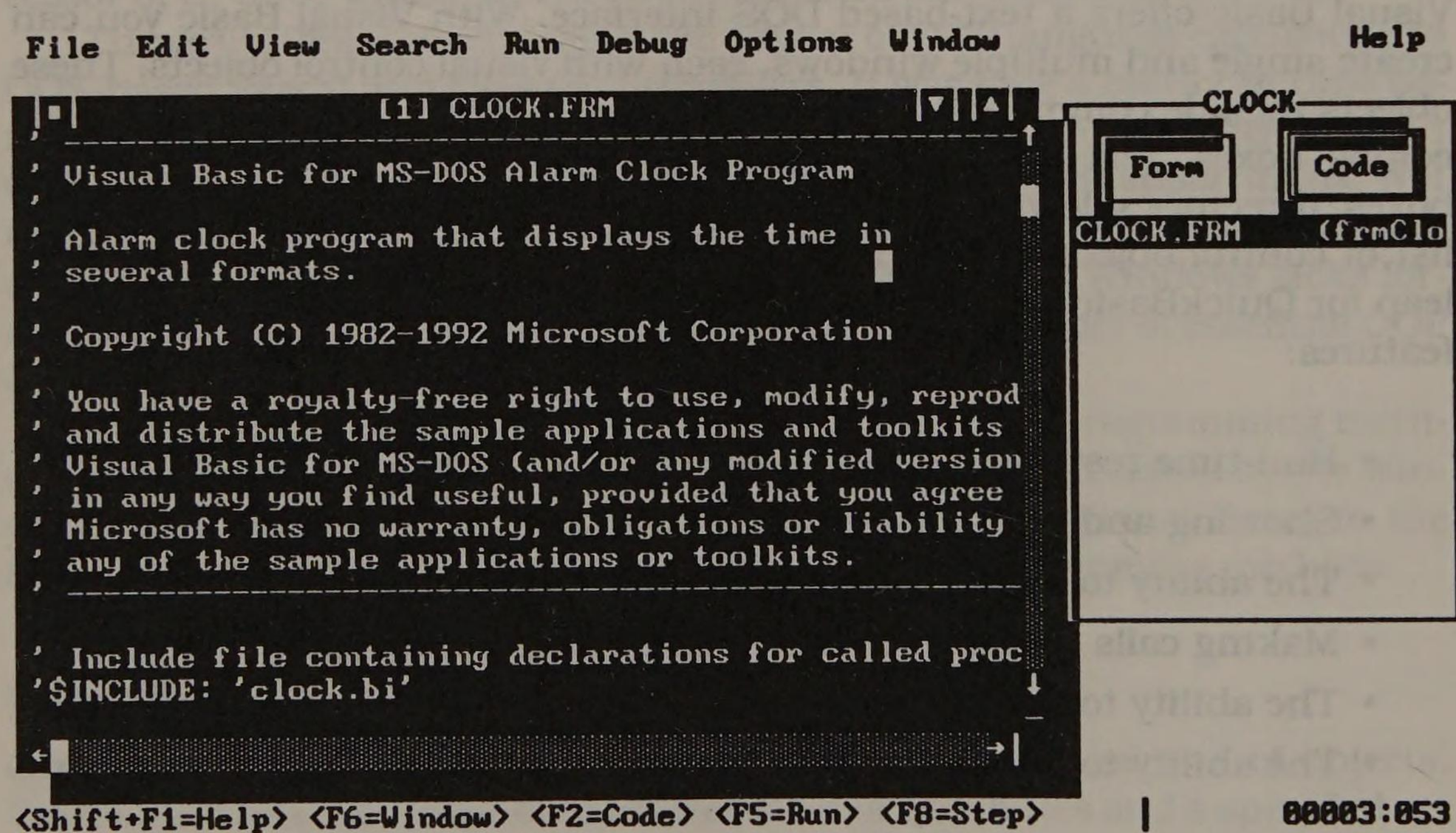
dio buttons appear in a frame control, only one can be selected at any time. Selecting another radio button in the same frame will automatically deselect other radio buttons.

Scroll bar A control that allows you to quickly select from a range of integers.

List box A control that allows you to select from a list of items.

Combo box A control that combines the text box and the list box controls. The combo box allows you to either select an item from the list box part or enter a new item in the text box part.

Timer A control that performs a certain action at regular time intervals.



1-1 The programming environment.

The Visual Basic control panel

Visual Basic provides sophisticated environments for developing professional looking DOS applications. These environments are the Programming and Form Design environments, shown in FIGS. 1-1 and 1-2, respectively.

Figure 1-1 reveals that the Programming environment includes a menu bar, status bar, and various windows. The menu bar contains the following menus and commands that manipulate programming a Visual Basic application.

The File menu Offers options to allow you to work with projects, forms, code modules, and files. The menu also offers options to print and spawn the DOS shell.

The Edit menu Presents options to cut, copy, paste, and delete text; create new SUBs and FUNCTIONs; and edit event procedures.

The View menu Contains options for viewing code, forms, include files, output screen.

The Search menu Offers options to find and replace text in the Code window.

The Run menu Allows you to run and restart programs; create .EXE or library files; select the start-up file; and modify the command-line arguments.

The Debug menu Presents options that allow you to manage watched variables and manipulate breakpoints.

The Options menu Permits you to set the display appearance of the Visual Basic environments, set the paths for several types of files, set the help path, toggle syntax checking, set the action of the right mouse button, and save the above options.

The Window menu Allows you to open and view the following windows:

- The Code window that displays the code attached to a form. Visual Basic allows you to view multiple Code windows.
- The Project window that contains the lists of files involved in the current programming project.
- The Output window that displays the output from a running program.
- The Immediate window that enables you to execute a single line containing Basic statements.
- The Help window that displays online help.
- The Calls window that shows the *call tree* which maps the sequence of routine calls.
- The Debug window that displays watched variables and breakpoints.

The Help menu Offers online help.

Figure 1-2 shows the Form Design environment which contains the menu bar, the properties bar, the Toolbox, the visual forms, and the color palette. The menu bar contains the following menu items:

The File menu Offers options to allow you to work with projects and forms. The menu also offers options to print and spawn the DOS shell.

The Edit menu Presents options to cut, copy, paste, and delete text, as well as edit event procedures.

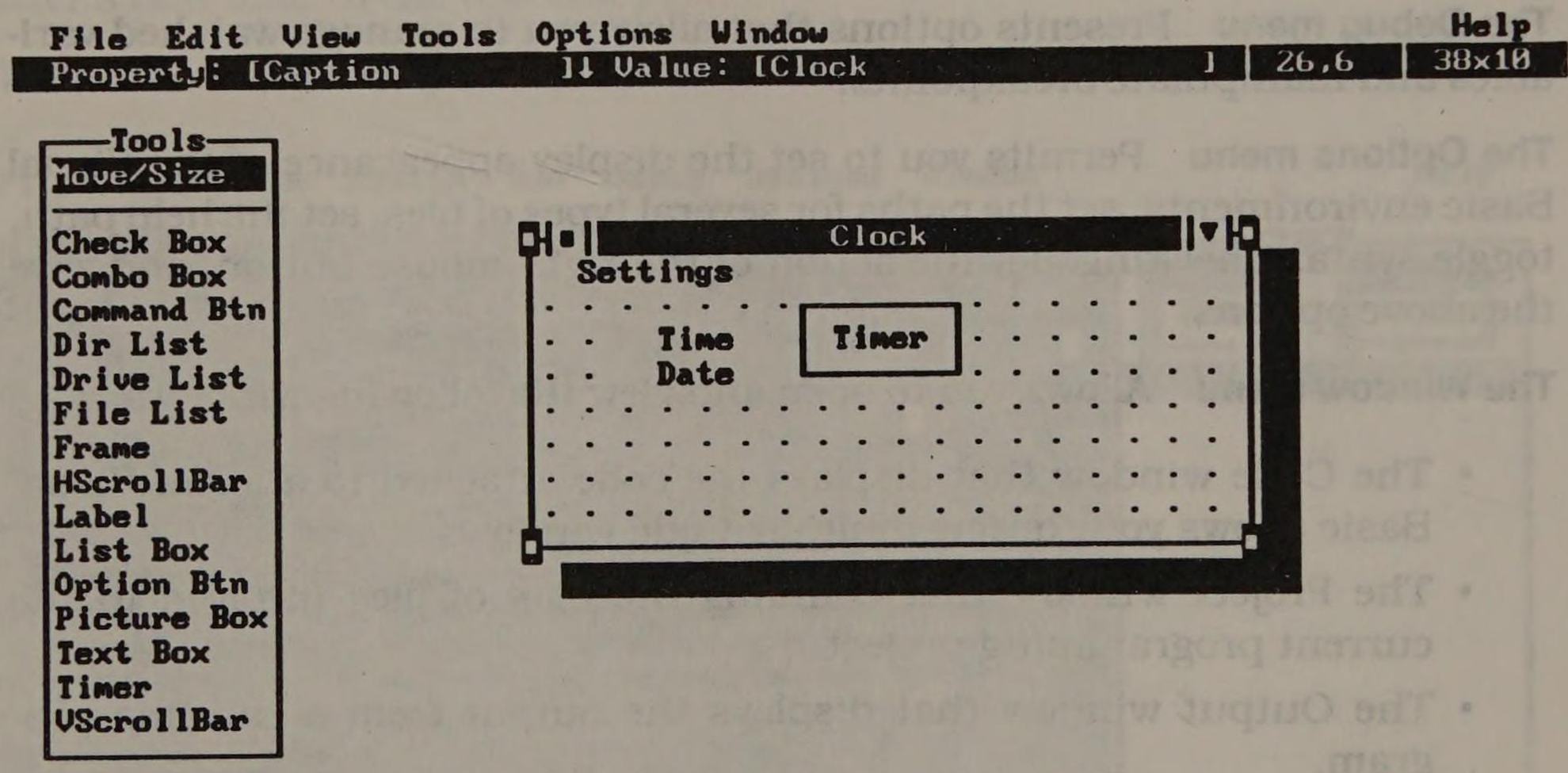
The View menu Contains options that allow you to view a Code window, a form, toggle the menu bar, toggle the properties bar, and toggle the grid lines used in designing forms.

The Tools menu Presents the tools that allow you to paste the visual controls in a form.

The Options menu Allows you to set the display appearance of the Visual Basic environments, set the help path, and save the above options.

The Windows menu Offers options to open the color palette window, the Toolbox, the menu design window, and the help window.

The Help menu Offers online help.



1-2 The form design environment.

Event-driven programming in Visual Basic

In an earlier section, I used the example of my CASIO watch to introduce you to the basic notion of event-driven programming. To explain more fully the dynamics of event-driven programming in Visual Basic, consider a typical company and its employees. Each employee has a title and a job description that defines his or her skills, responsibilities, and role in the company's daily activities. The company interacts with the outside world through a series of events—phone calls, mail, telexes, faxes, electronic bulletin boards, travels, meetings, etc. Each event is handled by the appropriate employee(s). For example, the incoming company mail is sorted and distributed by the company mailman. This is neither the job of the receptionist nor the accountant.

Just like the company's events, Visual Basic responds to the mouse and keyboard events by invoking the appropriate procedures. If none are found, the event is abandoned and ignored. The pseudocode that describes event-driven programming is typified by the following DO-LOOP and SELECT-CASE statements:

```
initialize_application
exit_flag = FALSE
DO
    GetNextEvent(NewEvent)
    SELECT CASE NewEvent
        CASE event1
            handle_event1
        CASE event2
            handle_event2
        CASE event3
            handle_event3
        CASE end_application
            handle_exit
            exit_flag = TRUE
        CASE ELSE
            ignore_event
    END SELECT
LOOP UNTIL exit_flag = TRUE
```

The above pseudocode shows that an event-driven application uses a loop to obtain the next event. Once the event is obtained, the SELECT-CASE statement is used to handle the event or ignore it (in the CASE ELSE clause). As the above pseudocode shows, there is usually a Case clause reserved for exiting the Do-Loop and ending the application.

The designers of Visual Basic have chosen to define a set of events for the form object and for each control object. The response of an object to an event is provided by a specific procedure. Visual Basic derives the name of the procedure from the names of the object and the event handled. The following general syntax is used:

SUB objectName_eventName(⟨optional list of parameters⟩)

Thus, the functionality of a form or a control object is defined by the associated event-handling procedures.

To give you a general idea about the life cycle of an event in Visual Basic, consider the event generated by clicking on a command button:

- The event is generated by clicking on a command button, titled Command1.
- The button (a control object) recognizes the action as an event that must be handled.
- Visual Basic looks for a procedure that matches the event and the object responding to it. The logic of a SELECT-CASE is used as shown in the pseudocode below:

```
SELECT CASE NewEvent
    ' click mouse button the Command1 button
    CASE Click
```

```
            Command1_Click()
    ' use tab key to select Command1 button
    CASE GotFocus
            Command1_GotFocus()
    ' press any key while Command1 button is selected
    CASE KeyPress
            Command1_KeyPress()
    CASE ELSE
            'ignore
END SELECT
```

Because the event is a mouse click, the event name is Click. Combining the event name with the button name, Command1, yields the name of the sought procedure, Command1_Click. This procedure is called to handle the click event. If the event is a key being pressed while the Command1 button was in focus (that is, selected), then the event name is KeyPress and the sought procedure is accordingly, Command1_KeyPress.

The properties of an object

I mentioned earlier that objects have characteristics and functionalities. The objectName_eventName procedures define the functionalities. What about the characteristics? The designers of Visual Basic have specified a fixed set of properties for each type of object. Thus, the form and control objects resemble user-defined (or factory-defined, if you prefer) types that specify the fields and their data types. Moreover, Visual Basic assigns default values (also called settings by the Visual Basic manuals) to each property. A good example is the properties of a text box. The following is a sample of four properties and their settings:

Property	Setting
CtlName	"Text1"
Text	"Text1"
ForColor	black
BackColor	white

The beauty of Visual Basic is that you can change the most settings during the application design time or at run-time. Changing the setting during design time is a manual process that involves clicking on the combo box in the Properties bar and typing new values. Changing the setting during run-time involves placing Basic statements that assign new setting values and/or modify the existing ones. When a setting is changed at run-time, Visual Basic updates the appearance of the object to reflect the new setting.

Visual Basic has a control name property, CtlName, for each control object. When a control is created, Visual Basic automatically assigns it a default control name. The default control name takes into consideration the order of creating that control. For example, the first text box has the default name Text1. The second text box has the control name of Text2, and so on.

You can and should change these default control names to better reflect the functionality of the control objects. In a similar way, Visual Basic uses a default text or caption for the form object and the control objects.

Starting Visual Basic

To start Visual Basic from DOS move to the directory containing Visual Basic and simply type VB at the DOS prompt. You can automate this task in a simple batch file that performs the above two steps.

To exit from Visual Basic, select the Exit option in the File menu of the Programming environment.

2

The Visual Basic Help system

Online help systems for programming languages have proliferated since 1989. In fact it was the Microsoft QuickBasic 4.5 compiler, which offered extensive online help that rivaled the product's own manuals. Microsoft has set the industry standard by applying hypertext technology and data compression. Visual Basic offers online help that is very similar to that of QuickBasic.

In this chapter, the online help system for Visual Basic is presented along with the various ways you can use it. It makes sense to get acquainted early on with the online help system, because you will most likely have a lot of questions to ask about diverse aspects of Visual Basic. This chapter covers these four aspects of the Visual Basic help system: it is context sensitive; it uses hypertext technology; it supports indexed searching for keywords; and it offers flexible navigation through the help information.

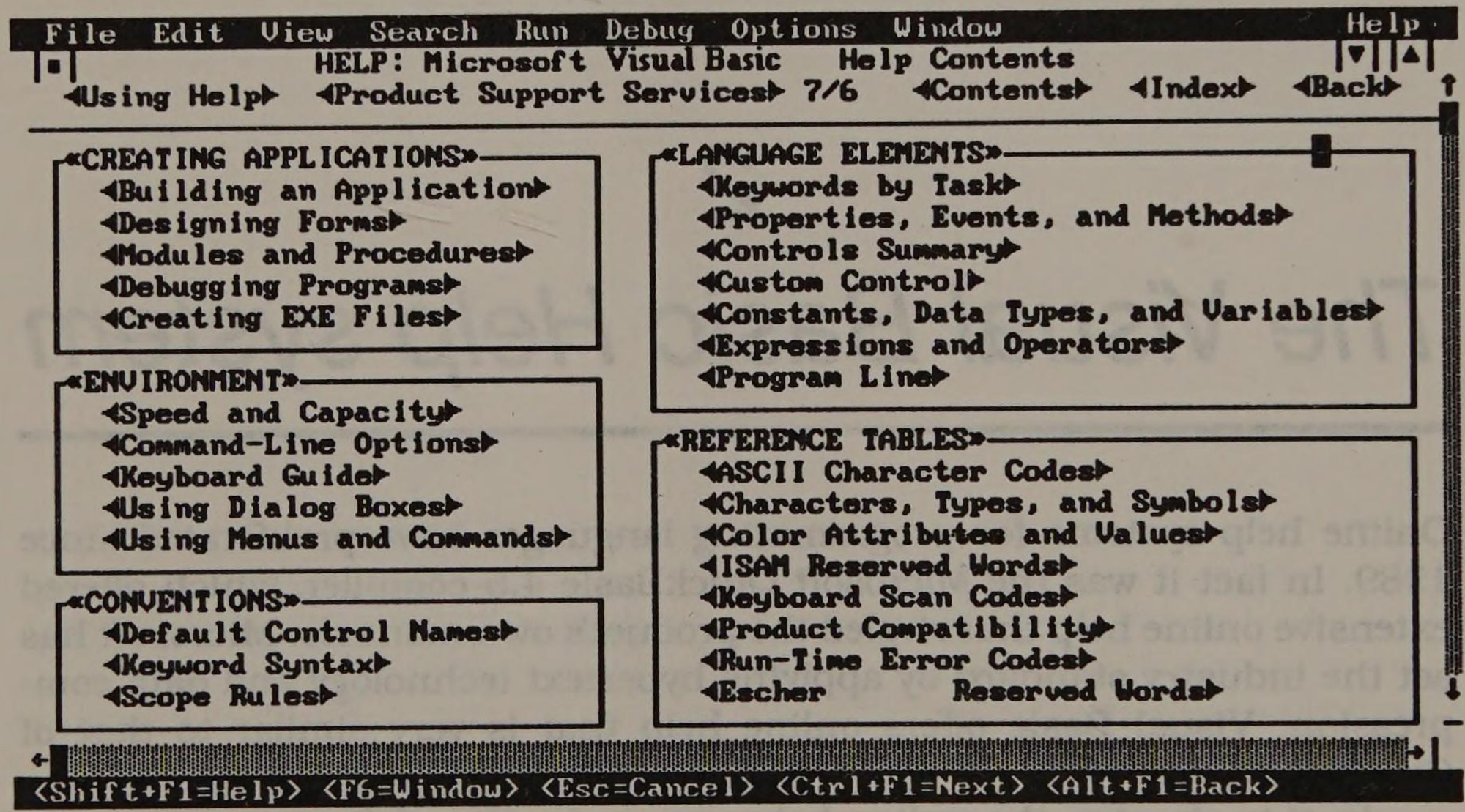
The Visual Basic help system is a potent tool at your fingertips. Do not shy away from using it. You can obtain information quickly and efficiently, because your computer will do the search for you.

The best way to learn about a new application is to use it, so I suggest that you start Visual Basic. Experiment with the features that I will be discussing in the upcoming sections.

About hypertext

If you are familiar with the hypertext technology, you can skip this section. Hypertext is a technique that links various parts of a usually long document using hypertext links. These links are special words that are part of the text, but are distinguished by having a different display attribute (usually a

catchy color) or are surrounded by special characters. Internally, these links are associated with pointers to other locations in the document. The links guide the hypertext viewer (the utility that permits you to view hypertext documents) in jumping to another part of the document associated with the hypertext link. Often, the hypertext viewer enables you to backtrack to the previous document locations; this is also the case with the Help system in Visual Basic.



2-1 The Contents help screen.

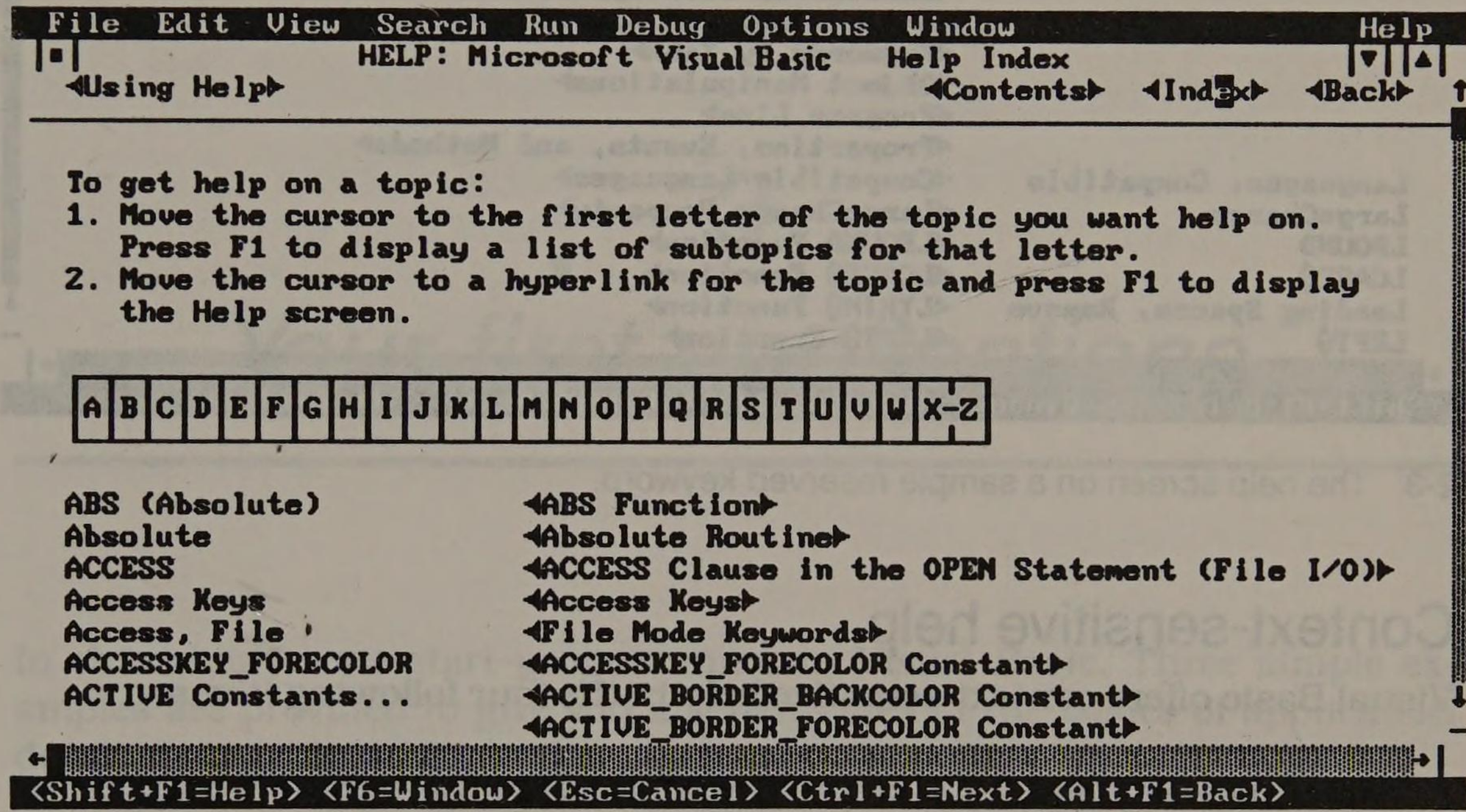
Navigating through the Help system

When you invoke Visual Basic, select the Help menu in the menu bar of the Programming environment. The Help menu options include the following:

The Contents option Allows you to view the general topics of help, as shown In FIG. 2-1. The Contents help screen offers the following general areas:

- The Creating Applications topics: these include designing the forms, writing the modules and procedures, debugging programs, and creating .EXE files.
- The Language Elements topics: these include keywords (by task), properties, events, methods, controls, constants, data types, variables, operators, expressions, and program lines.
- The Environment topics: these include the command-line options, keyboard guide, using help, using dialog boxes, using menus, and using commands.

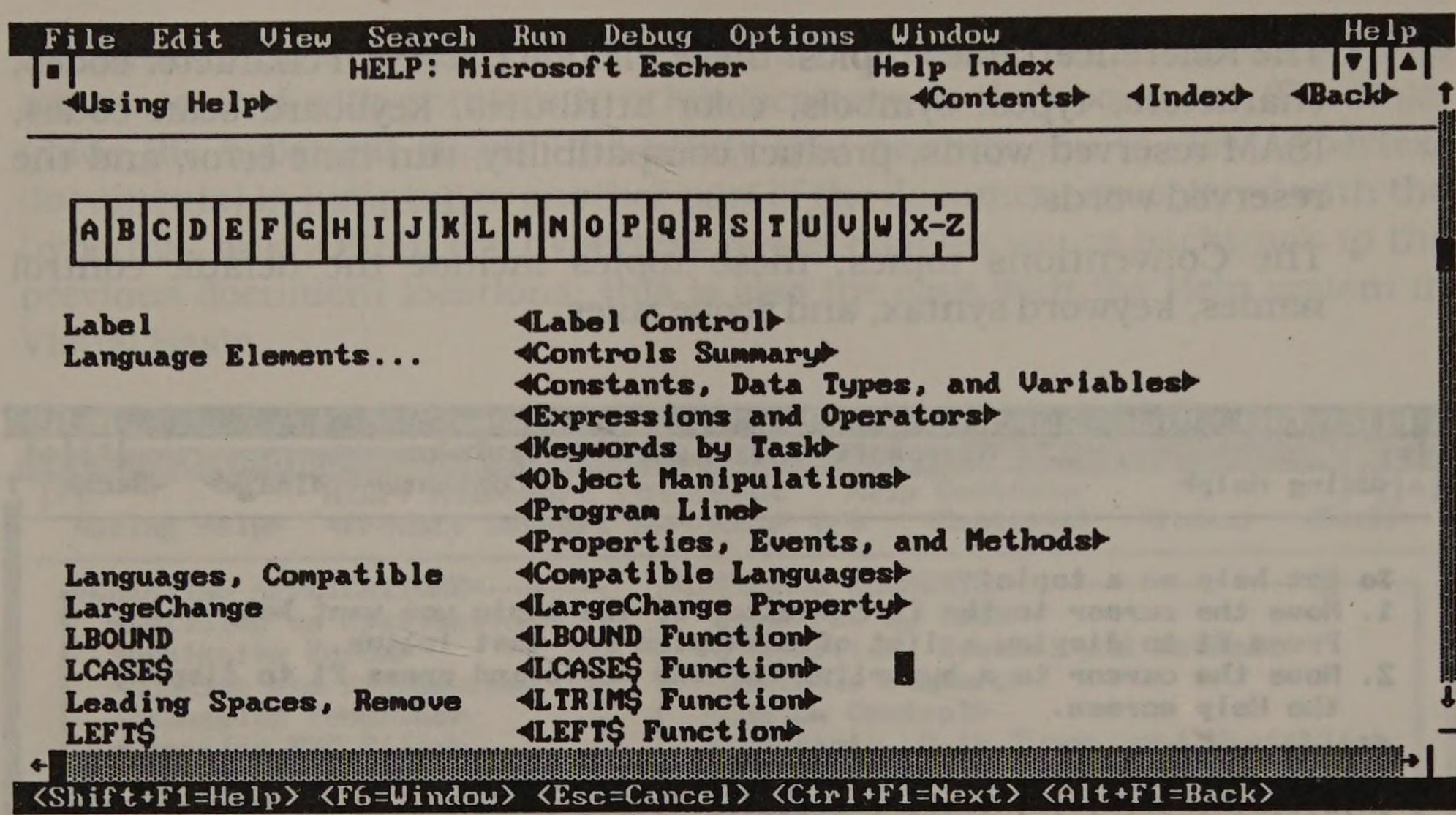
- The Reference Tables topics: these include the ASCII character codes, characters, types, symbols, color attributes, keyboard scan codes, ISAM reserved words, product compatibility, run-time error, and the reserved words.
- The Conventions topics: these topics include the default control names, keyword syntax, and scope rules.



2-2 The main Index help screen.

The Index option Allows you to select a topic by index. This type of help first displays the general index, shown in FIG. 2-2. This screen allows you to select a section by moving the cursor to an indexing letter and then pressing the F1 function key. Figure 2-3 shows a sample help on a reserved keyword. The scrollable indexed screens display the Contents, Index, and Back hypertext links in their upper right area. The active hypertext links in Visual Basic are enclosed by thick green arrows. The inactive hypertext links appear surrounded by thick white arrows. As you move to other help screens, other hypertext links may appear in the upper-left area or even inside the help text. When you select a reserved keyword, the help system displays the Summary, Details, and Example hypertext links in the upper-left side of the help screen.

The Summary hypertext link leads to the syntax description of the Visual Basic keyword. The Details hypertext link offers additional details on using the reserved keyword. The Example keyword displays a short sample code. You can copy that code (or any parts of it) by highlighting the parts to copy, pressing the CTRL + INS, and then pasting it in a Code window by pressing the SHIFT + INS keys.



2-3 The help screen on a sample reserved keyword.

Context-sensitive help

Visual Basic offers context-sensitive help in the four following items:

- To receive help on a menu command highlight that command by pressing the ALT key and then using the direction key, then press the F1 function key.
- To obtain help on a reserved keyword in a Code window, move the cursor to that keyword and press the F1 function key. The Visual Basic help system displays the online help text related to that keyword.
- To get help on a dialog box, either click on the Help button in the dialog box or press the F1 function key.
- To get help on an error message, either click on the Help button in the error message dialog box or press the F1 function key.

Additional help

If you are somewhat overwhelmed by the Help system itself, there is some good news. The Help system offers information on itself. This is found in the Help menu option of the help screens. The same menu option also has an online tutorial option.

3

Your first applications

In this chapter we start programming in Visual Basic. Three simple examples are provided to give you a general idea of the basics of application development. This chapter discusses:

- The basic steps involved in creating a Visual Basic application.
- The Project window.
- The project files.
- Working with a form and with control objects.
- Setting the properties of the control objects.
- Attaching code to control objects.
- Running and halting a Visual Basic application.
- Saving and loading Visual Basic project files.

☞ In this book, I assume that all of the Visual Basic application files are stored in the directory \VBDOS\EASYVB. A separate directory prevents mixing the Visual Basic files of this book with other Visual Basic files that you might have already developed or obtained from a friend. Please create this directory now before you proceed with the examples.

The programming examples in this chapter expose you to the three basic steps involved in crafting a Visual Basic application. The next three chapters examine in more detail each of these steps.

The first Visual Basic application

The first Visual Basic application performs the following simple task: click on a button to view the current date and time. Simple, but how do we get there? The first step is to load Visual Basic, as explained in chapter 1. When Visual Basic is loaded you find yourself in the Programming environment. You need to move to the Form Design environment. However, because you are starting a new project, you need to open a new form. Select the New Form option in the File menu. Visual Basic uses a dialog box to prompt you for a form name. Select the EASYVB directory if it is not already the current directory. Enter APP1 and Visual Basic will automatically append the .FRM extension to your input.

The Form Design environment systematically puts an empty form window on the screen. This form is part of a new Visual Basic application project. The form window represents a stripped-down window with minimal functionality. To build a Visual Basic application you need to perform the following basic steps:

1. Construct the interface for your application by drawing the required control objects on the form.
2. Set (or fine-tune, if you prefer) the properties of the form and controls to influence their physical appearance and behavior.
3. Attach Basic code to the form and/or control objects.

Visual Basic projects

The above steps are performed in the framework of a project. To better explain a Visual Basic project, bring up the Project window (if it is not already displayed) in the Programming environment. To display the sought window select the Project window option from the Window menu. The project window lists the files involved in a Visual Basic application. There are two types of such files:

Form files Contain the interface parts of a Visual Basic application including:

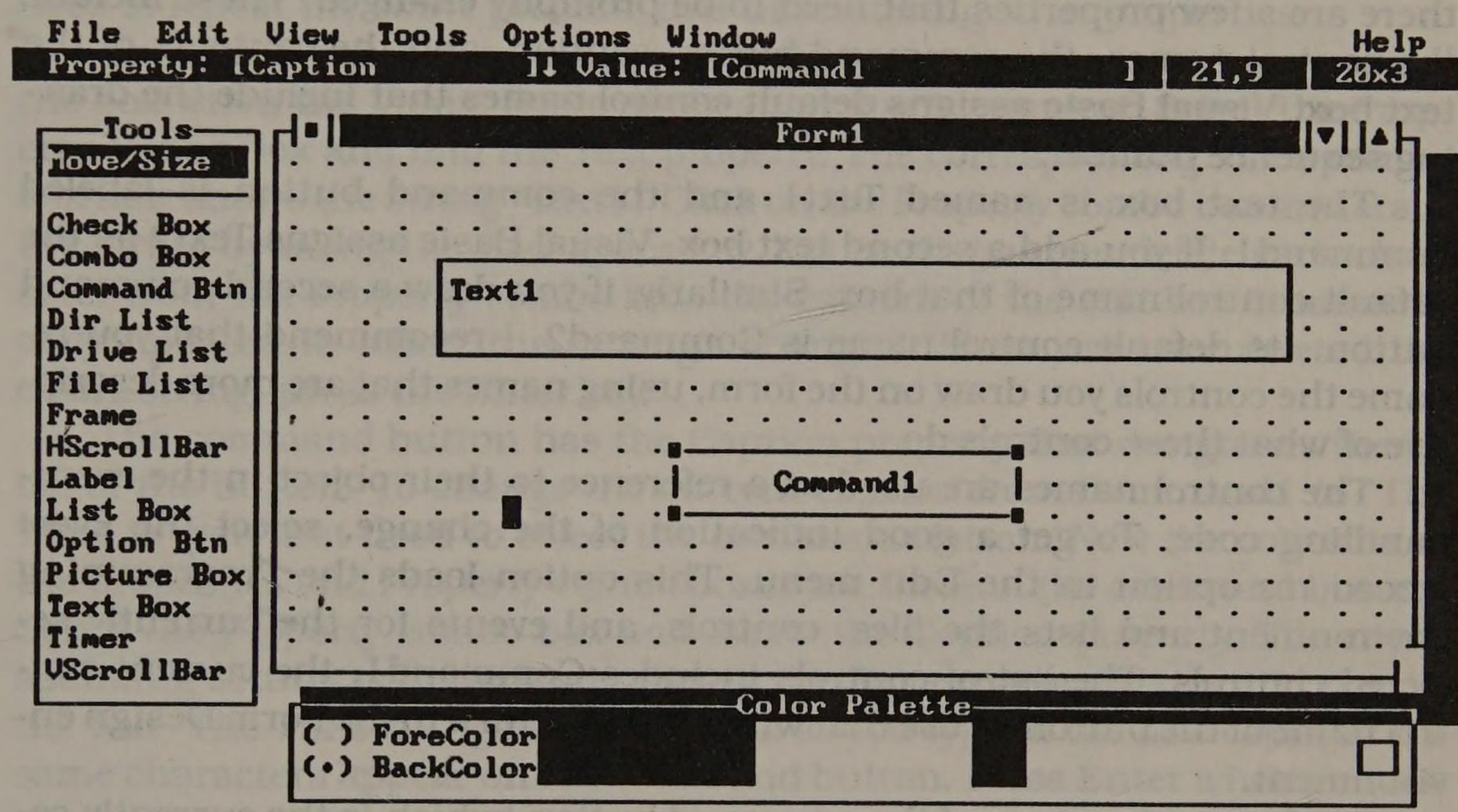
- The controls drawn on each form.
- The event-handling procedures attached to the forms and controls.
- The general definitions and procedures that are not attached to the forms or control objects.
- Included Basic files. These files have the .BI extension and are incorporated in a form using the INCLUDE metacommand whose general form is shown below:

```
'$INCLUDE: 'filename.bi'
```

Form files typically have the .FRM extensions. You need at least one form in your Visual Basic applications.

Zero or more module files These are .BAS files that contain libraries of auxiliary and supporting procedures that are called by other parts of the application (other modules of the code attached in the forms and controls). Modules are suitable for long applications.

The project files are stored with the .MAK extension name.



3-1 The form of the first application with the text box and the command button.

Drawing controls in the form

The control objects are vital components of a Visual Basic application. The designers of Visual Basic have defined a specific set of controls. Each control has its preset collection of properties, default values assigned to these properties, behavior, and events that it can handle. These predefined items greatly reduce the amount of coding you need to perform, and still offer flexible control over your application design.

To draw the controls on the form you need to invoke the Toolbox window by selecting the Toolbox option in the Window menu. The Toolbox contains a list of controls. To select a control, click on its name in the Toolbox and then move the cursor to the form window. Holding the mouse button down and dragging the mouse cursor draws the selected control in the form.

Now let's start drawing the controls needed by the application. First, select the text box from the Toolbox and draw a rather wide text box on the form (see FIG. 3-1). Next, select the command button from the Toolbox and draw a command button (see FIG. 3-1). The command button is now the cur-

rently selected control, as indicated by the black square dots that appear at the edge of the control.

This ends the first basic step in developing a Visual Basic application.

Altering the control properties

Visual Basic assigns default names to the properties of the controls you draw. While most of these default values are appropriate, at least for now, there are a few properties that need to be promptly changed. These include the control names, the command button caption, and the contents of the text box. Visual Basic assigns default control names that include the drawing sequence number.

The text box is named Text1 and the command button is labeled Command1. If you add a second text box, Visual Basic assigns Text2 as the default control name of that box. Similarly, if you draw a second command button, its default control name is Command2. I recommend that you rename the controls you draw on the form, using names that are more descriptive of what these controls do.

The control names are used as a reference to their object in the event-handling code. To get a good indication of the change, select the Event Procedures option in the Edit menu. This option loads the Programming environment and lists the files, controls, and events for the currently selected controls. The list of controls includes Command1, the current control name of the button. Close that window and return to the Form Design environment.

To change the name of the command button (which is the currently selected control) from Command1 to Button, perform the following steps:

1. Scroll though the Property combo box, located in the Properties bar, until you find CtlName (the properties are sorted alphabetically). As you scroll through the property list, notice that the corresponding property value in the neighboring Property combo box also changes.
2. The value of CtlName is Command1. Click on the Value combo box and type in Button. This changes the name of the command button. Once you are done keying in the new control name, press the Enter key. To cancel what you have typed in and resort to the original value, press the ESC key.

   The new control name of the command button should be reflected in the event procedure window. Select the Event Procedures option in the Edit menu to view the list of files, objects, and events. The name of the button control appears as Button in the Object list, an indication that Visual Basic has taken into account the new control name of the command button. Return to the Form Design environment.

Warning! Always change the control names *before* you start attaching code to these controls. Failing to do so causes Visual Basic to hide the original code and to present you with a new empty procedure.

Now select the text box and perform a similar operation to change the control name from Text1 to OutputLine. Because control names are not involved in the physical appearance of the control objects, you will not see any changes in the form; the text box still contains the string "Text1" and the command button is still labeled Command1. To change these settings you need to repeat the above process, this time altering other properties.

The text box has the Text property that specifies the initial contents of the box when the Visual Basic application starts running. Select the Property combo box and find the Text property. The corresponding Value combo box now shows the string "Text1." Click on the Property combo box and erase the string by using the Backspace key. Notice that as you delete each character from the Property combo box, the content of the text box is immediately updated to reflect the setting's changes. When you have deleted the entire string, press the Enter key.

The command button has the Caption property that designates the label of the button. To change the current caption from Command1 to "The Time is" you first need to select the command button. Now the contents of the properties and Property combo boxes are those of the command button. Select the Property combo box and locate the Caption property. The corresponding setting is Command1. Select the Property combo box and type in the text "The Time is." Again, notice that as you type in the new caption, the same characters appear on the command button. Press Enter when you finish keying in the new caption. Figure 3-2 shows the updated form.

Attaching event-handling procedures

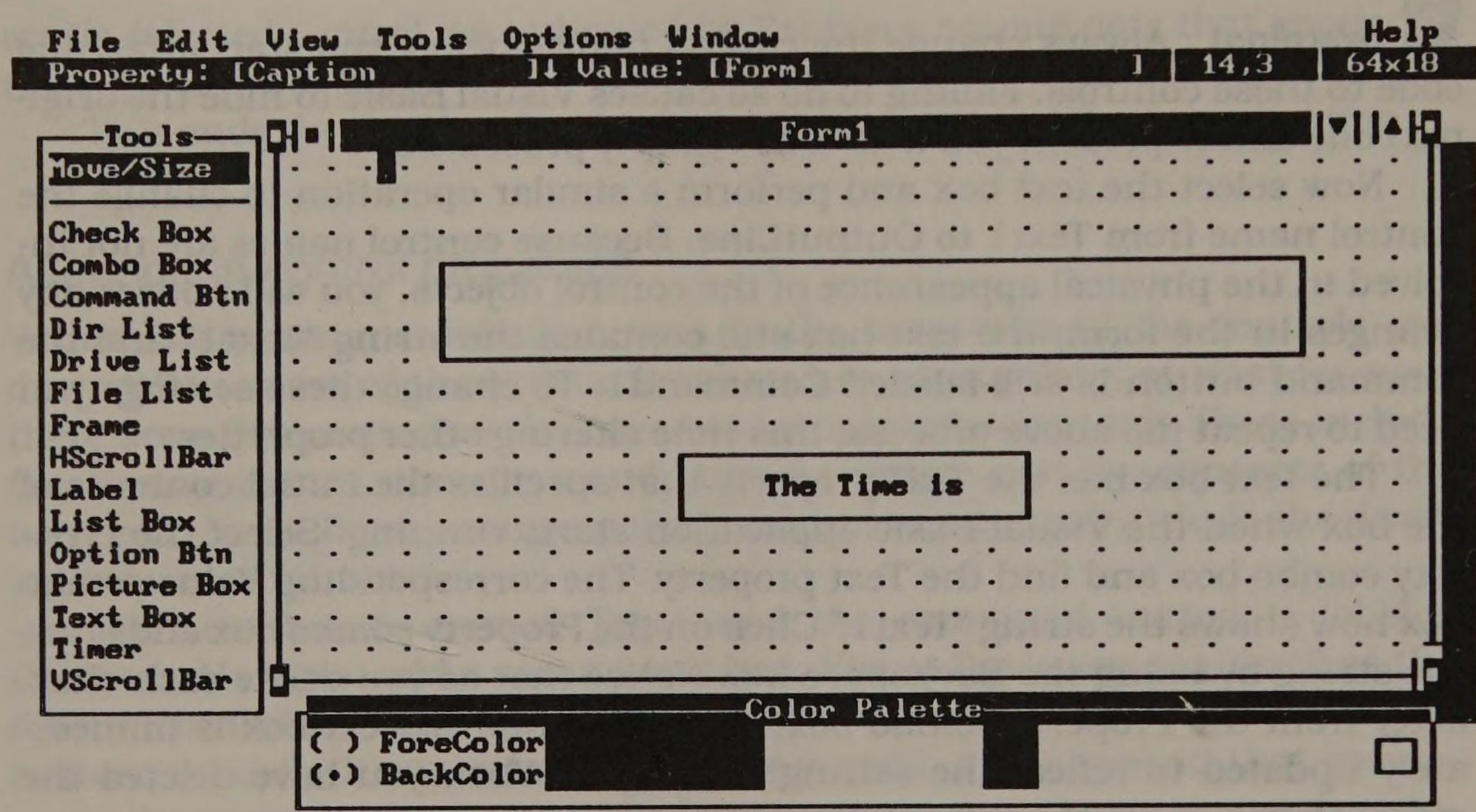
So far, the form provides the sought visual appearance, but is otherwise a dud. This is due to the lack of event-handling procedures that define the exact response of the controls to user-generated events. The designers of Visual Basic offer a predefined set of events for each class of controls. This greatly simplifies things for you the programmer; it is a matter of selecting which control responds to which event. In the case of the first application, we need to make the command button respond to mouse clicks.

To attach the needed code, you need to perform the following steps:

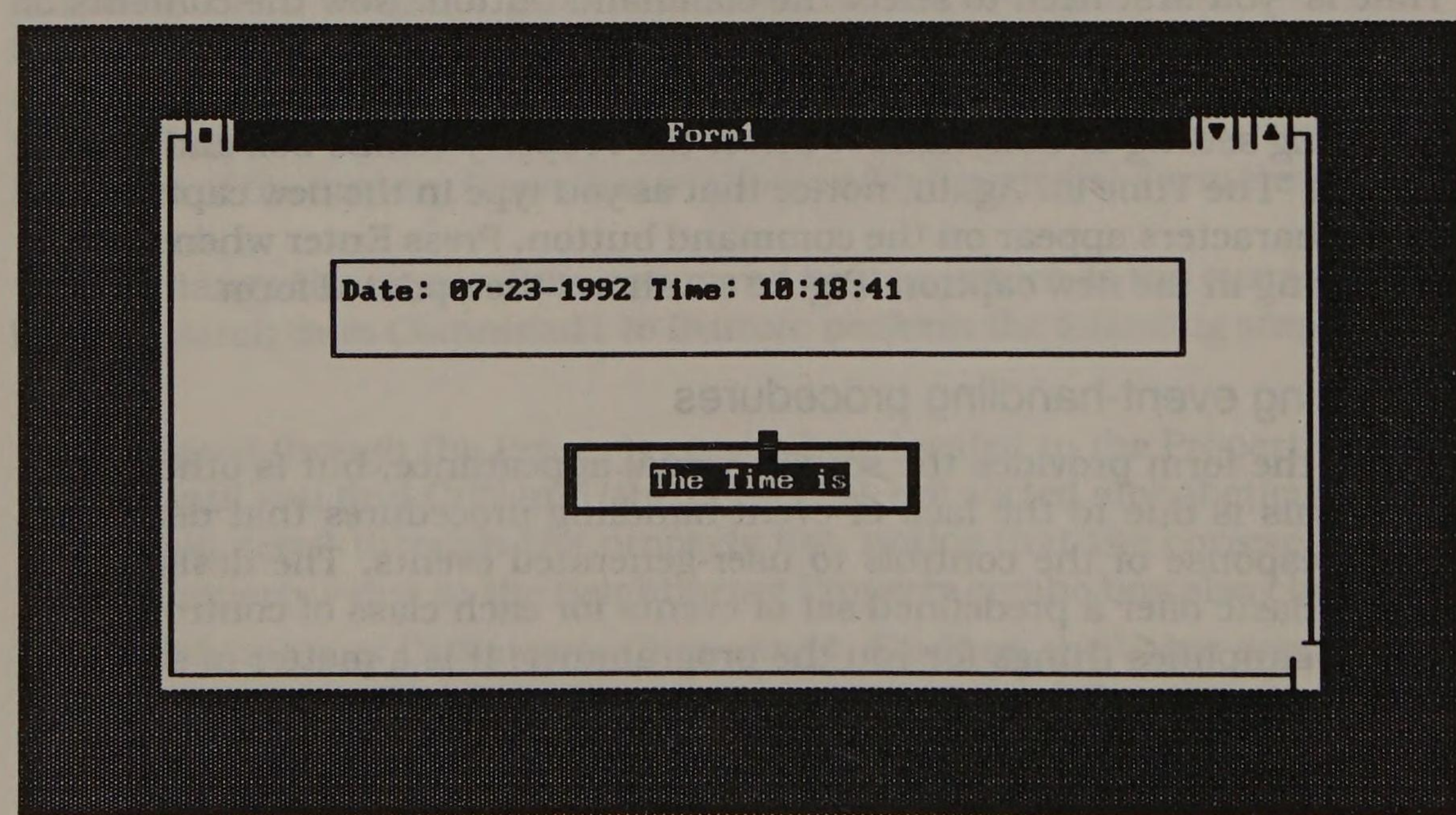
1. Select the Event Procedures option in the Edit menu.
2. Select the Button object in the Objects list.
3. Select the item Click in the Events list.

The previous steps bring up the Code window with the empty procedure Button_Click(). Type in the following statement inside the procedure:

```
OutputLine.Text = "Date:" + DATE$ + "Time:" + TIME$
```



3-2 The updated form for the first application.



3-3 The first application run-time window.

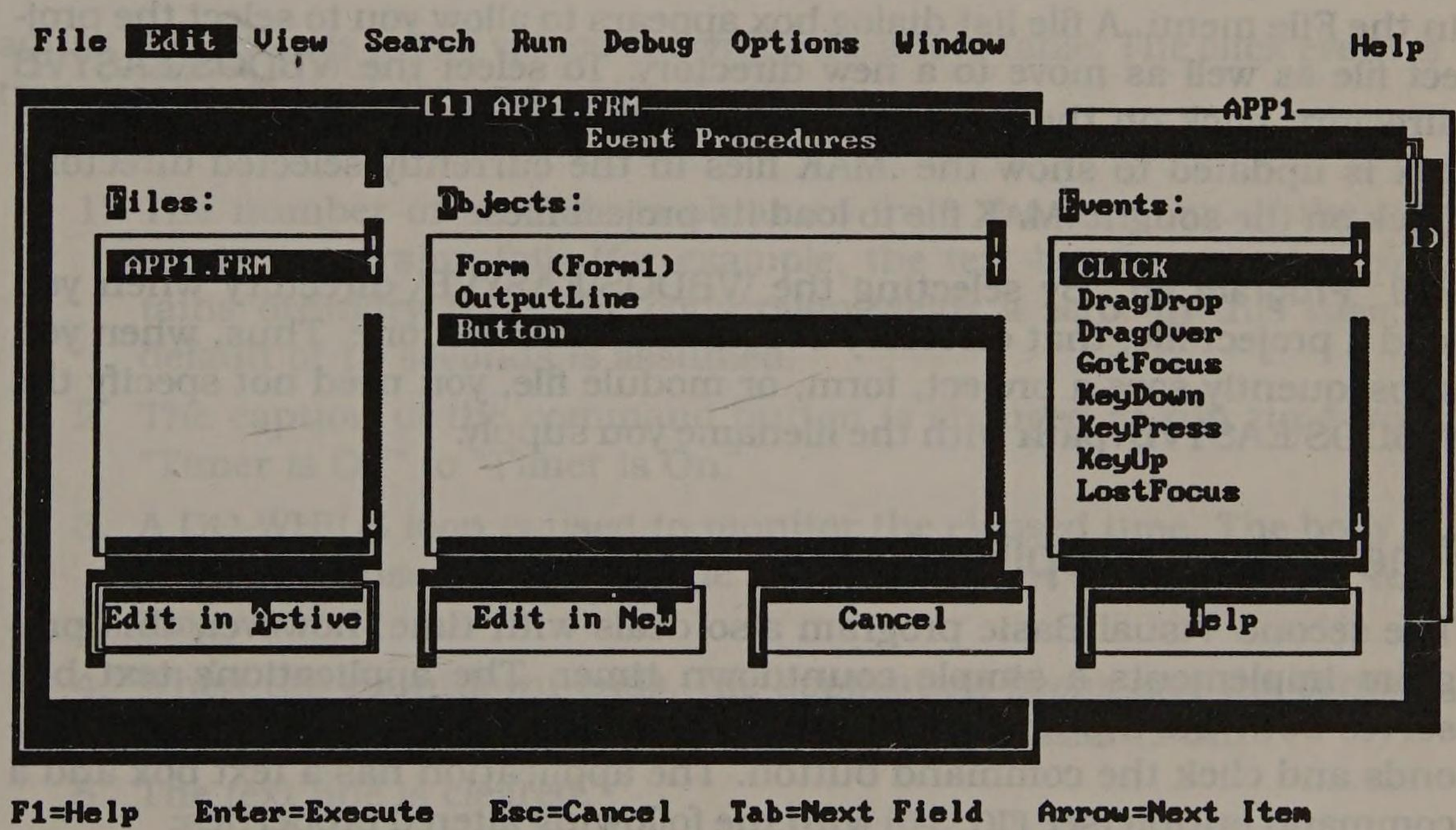
The statement you type assigns a string expression that contains the system date and time to the Text property of the OutPutLine text box. The general syntax for assigning a new setting to a property of the form and other control objects is:

```
objectName.propertyName = newSetting
```

Likewise, the general expression objectName.propertyName can appear on the right-hand side of the equal sign to retrieve an object's property. The syntax resembles that of manipulating the fields of user-defined data types.

Every time you click on the command button, the text box displays the current date and time. To run the program, press the F5 function key from the Programming environment. If you have made a syntactic error in typing the previous Basic statement, the Visual Basic compiler will indicate your mistake using an error window. If the program is correct, it generates a screen image that should resemble FIG 3-3. You can click on the buttons as many times as you like. To stop the application, press the CRTL+BREAK keys, or CTRL+F4 keys.

When you bring up the Event Procedures window, Visual Basic displays lists that show the project's files, the objects in the currently selected file, and the events that are available to the currently selected object. Visual Basic displays the events that have a handler in uppercase. Therefore, you can tell at a glance which events are handled by the currently selected object. Figure 3-4 shows the Event Procedures window that contains the various objects and events.



3-4 The Event Procedures window showing the various objects and events.

Saving your project

Saving the modified project and program files is a practical part of programming in any language, including Visual Basic. Select the Save Project option in the File menu to open a dialog box that allows you to specify the name of the project's file. Select the EASYVB directory if it is not already the current

directory. Enter APP1 and Visual Basic automatically appends the .MAK extension to your input.

You also can save the code attached to the currently selected form or module to an ASCII text file. Make the form the currently selected object, and choose the Save Text option in the File menu. A file list dialog box appears to let you specify the target filename. Type in APP1.TXT to store the code of the form in file APP1.TXT.

Warning! Do not rename or delete the project files outside Visual Basic. This puts the project file out of sync with the actual list of files, which leads Visual Basic to display an error message to warn you that some of the project files are missing.

Always update and manage the project files from within Visual Basic to ensure that the project file is correctly keeping track of the application files.

Opening your project

When you load Visual Basic, a new empty project is placed in the memory space. There is always one project loaded in the working memory of Visual Basic at any one time. To load your project files, select the Open Project option in the File menu. A file list dialog box appears to allow you to select the project file as well as move to a new directory. To select the \VBDOS\EASYVB directory, click on the EASYVB entry in the directory list box. The file list box is updated to show the .MAK files in the currently selected directory. Click on the sought .MAK file to load its project files.

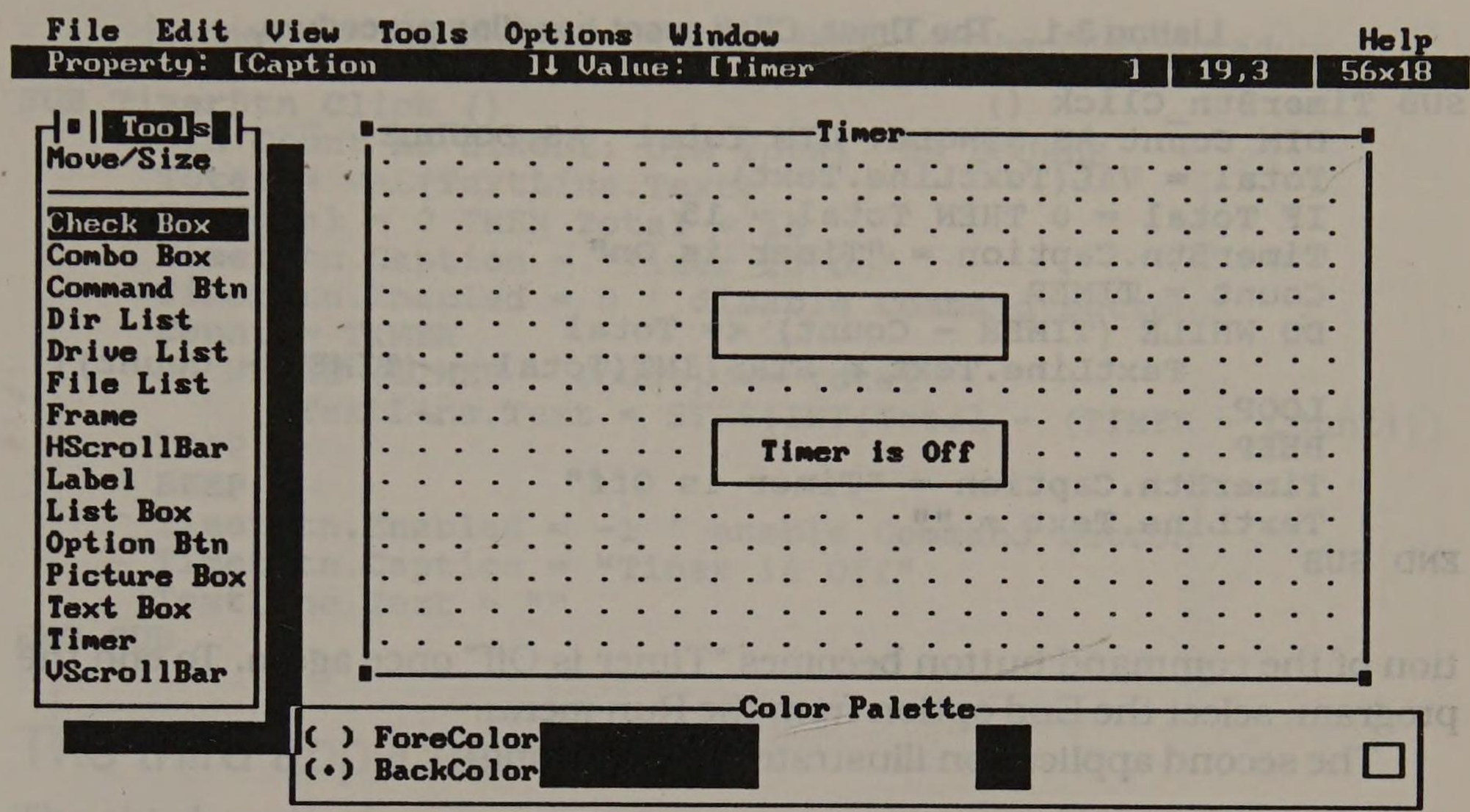
Program tip By selecting the \VBDOS\EASYVB\ directory when you load a project file, that directory becomes the current one. Thus, when you subsequently save a project, form, or module file, you need not specify the \VBDOS\EASYVB\ path with the filename you supply.

The second application

The second Visual Basic program also deals with time; however, this program implements a simple countdown timer. The application's text box serves both the input and output. You type in the start-up number of seconds and click the command button. The application has a text box and a command button (see FIG. 3-5) with the following altered properties:

Object	Property	Custom Setting
Form	Caption	Count Down Timer
Text Box	CtlName	TextLine
	Text	(empty string)
Command Button	CtlName	TimerBtn
	Caption	Timer is Off

Save the form and the project files in the APP2.FRM and APP2.MAK, respectively. Draw the Text box and command button, as described earlier, and



3-5 The updated form for the second Visual Basic application.

adjust the settings to the values shown in the above table. The click event is translated into the following series of actions:

1. The number of seconds is obtained from the text box. If the numeric conversion fails (for example, the text box is empty or contains ordinary text), the conversion yields a zero. In this case, a default of 15 seconds is assumed.
2. The caption of the command button is changed at run time from "Timer is Off" to "Timer is On."
3. A DO-WHILE loop is used to monitor the elapsed time. The body of the loop is used to update the countdown that appears in the text box.
4. When the countdown ends, the application beeps and the caption of the command button is changed back to "Timer is Off."
5. The text line is cleared.

Listing 3-1 contains the code of the procedure TimerBtn_Click that handles the click events for the command button.

Run the application by pressing the F5 function key while in the Programming environment. To use the countdown timer, select the text box and type in the maximum number of seconds—10, for example. Click on the "Timer is Off" command button and watch the text box display the countdown in seconds. While the countdown is in progress the command button has the caption of "Timer is On." When the countdown ends, the cap-

Listing 3-1 The Timer_Click event-handling procedure.

```
SUB TimerBtn_Click ()
    DIM Count AS SINGLE: DIM Total  AS DOUBLE
    Total = VAL(TextLine.Text)
    IF Total = 0 THEN Total = 15
    TimerBtn.Caption = "Timer is On"
    Count = TIMER
    DO WHILE (TIMER - Count) <= Total
        TextLine.Text = STR$(INT(Total - (TIMER - Count)))
    LOOP
    BEEP
    TimerBtn.Caption = "Timer is Off"
    TextLine.Text = ""
END SUB
```

tion of the command button becomes "Timer is Off" once again. To end the program, select the End option from the Run menu.

The second application illustrates the following:

- Using the text box for input and output. The expression VAL(TextLine.Text) obtains the number that you key in the text Box, the input. The assignments made to TextLine.Text update the caption of the text box to show a new output.
- Changing the setting of a control object at run time. The assignments made to TimerBtn.Caption altered the caption of the command button while the Visual Basic application is executing.

The changes made to the command button might include more profound ones, such as disabling the button. When the code of Listing 3-1 is running, the caption of the command button changes, but it might not be clear to a user whether or not the button is still active. The answer is that the button is effectively disabled while the attached code is running. To better reflect this fact, you can insert the following statement:

```
TimerBtn.Enabled = 0 ' disable Command button
```

after the statement that assigns "Timer is On" to the command button caption. You also need to insert the following statement after the Beep command:

```
TimerBtn.Enabled = 1 ' disable Command button
```

Now run the application by pressing the F5 function key. Click on the command button. What do you see? The button not only changed the caption, but the appearance of the caption text has changed. It is now a shaded text, indicating that the button is disabled. When the execution of the attached code ends, the button is enabled once more. This is reflected by the change in the appearance of the caption. To end the program, select the End option from the Run menu.

Listing 3-2 The modified TimerBtn_Click event-handling procedure.

```
SUB TimerBtn_Click ()
      DIM Count AS SINGLE: DIM Total  AS DOUBLE
      Total = VAL(TextLine.Text)
      IF Total = 0 THEN Total = 15
      TimerBtn.Caption = "Timer is On"
      TimerBtn.Enabled = 0 ' disable Command button
      Count = TIMER
      DO WHILE (TIMER - Count) <= Total
            TextLine.Text = STR$(INT(Total - (TIMER - Count)))
      LOOP
      BEEP
      TimerBtn.Enabled = -1 ' enable Command button
      TimerBtn.Caption = "Timer is Off"
      TextLine.Text = ""
END SUB
```

The third application

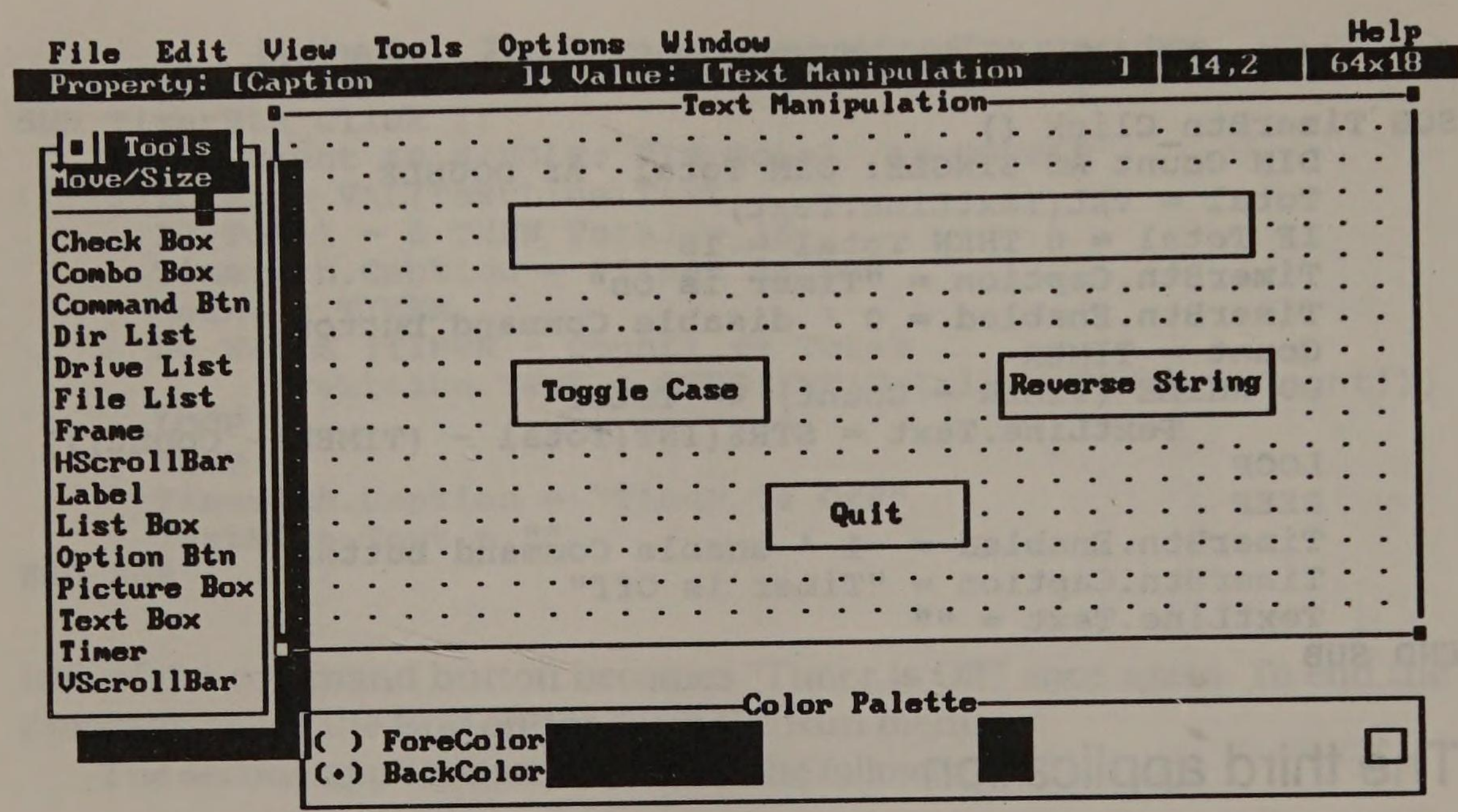
The third application is a simple window with a text box and three command buttons. One command button toggles the character case of the string in the text box. Another command button reverses the characters of the string in the text box. The third button ends the program execution. The following table lists the control objects and indicates the program's custom settings:

Object	Property	Custom setting
Form	Caption	Text Manipulation
Text Box	CtlName	TextLine
Text	(empty string)	
Command Button 1	CtlName	ToggleBtn
Caption	Toggle Case	
Command Button 2	CtlName	ReverseBtn
Caption	Reverse String	
Command Button 3	CtlName	QuitBtn
Caption	Quit	

Save the form and the project files in APP3.FRM and APP3.MAK, respectively. Figure 3-6 shows the application form with its control objects, after customization.

The text manipulation program works by having the various command buttons perform the following tasks:

The ToggleBtn command button The procedure attached to this button toggles the character case of the string in the text box. If the text box is empty, the procedure simply assigns "Hello World!" If the text box is not empty, the procedure performs these steps:



3-6 The updated form of the third Visual Basic application.

1. If the uppercase flag is 0 then assign 1 to the uppercase flag, convert the characters of the text box into uppercase.
2. Else assign 0 to the uppercase flag, convert the characters of the text box into lowercase.

The ReverseBtn command button The procedure attached to this button reverses the characters of the text box. If the text box is empty, the procedure simply assigns the string "Hello World!" Otherwise, it performs the following steps:

1. Assigns the length of the text box string to the local variable L.
2. Assigns a null string to the local variable S.
3. Uses a FOR-NEXT loop to copy the characters of the text box to string S, in reverse.
4. Assigns the string S to the text box.

The QuitBtn command button The code attached to this button simply ends the program execution with an End statement.

Listing 3-3 contains the code for the three procedures. Notice that the ToggleBtn_Click() procedure declares the local static variable Is_UCase. When the application starts to execute, this variable is assigned a 0. Consequently, the first time you press the Toggle command button (and the text box is not empty), the string in the text box is converted into uppercase. The second time you click on the Toggle Case button, the text is converted into lowercase, and so on.

Listing 3-3 The code attached to the various controls of the text-manipulating program.

```
SUB QuitButton_Click ()
    END
END SUB

SUB ReverseButton_Click ()
      DIM S AS STRING
      DIM I AS SINGLE: DIM L  AS INTEGER
      IF TextBox.Text = "" THEN
          TextBox.Text = "Hello World!"
      ELSE
          L = LEN(TextBox.Text)
          S = ""
          FOR I = 1 TO L
              S = S + MID$(TextBox.Text, L + 1 - I, 1)
          NEXT I
          TextBox.Text = S
      END IF
END SUB

SUB ToggleButton_Click ()
    STATIC Is_UCase AS INTEGER
    IF TextBox.Text = "" THEN
        TextBox.Text = "Hello World!"
    ELSE
        IF Is_UCase = 0 THEN
            Is_UCase = 1
            TextBox.Text = UCASE$(TextBox.Text)
        ELSE
            Is_UCase = 0
            TextBox.Text = LCASE$(TextBox.Text)
        END IF
    END IF
END SUB
```

Run the program by pressing the F5 function key. You can simply click on the Toggle and Reverse command buttons and watch the manipulation of the string "Hello World!" Click on the Quit button whenever you wish to end the program execution. This takes you back to the windows that were visible right before you started running the program.

4

Drawing the application form

This chapter focuses on the first main step in developing a Visual Basic application, namely, the design of the visual interface. In the last chapter I presented an overview of this step while developing the first application. This chapter digs a bit deeper into the topic.

The design of the visual interface for your Visual Basic application involves drawing the various control objects in the application's form(s). You should not rush into pasting various controls on the form. Instead, apply the general disciplines of good programming practices and plan ahead using the following steps:

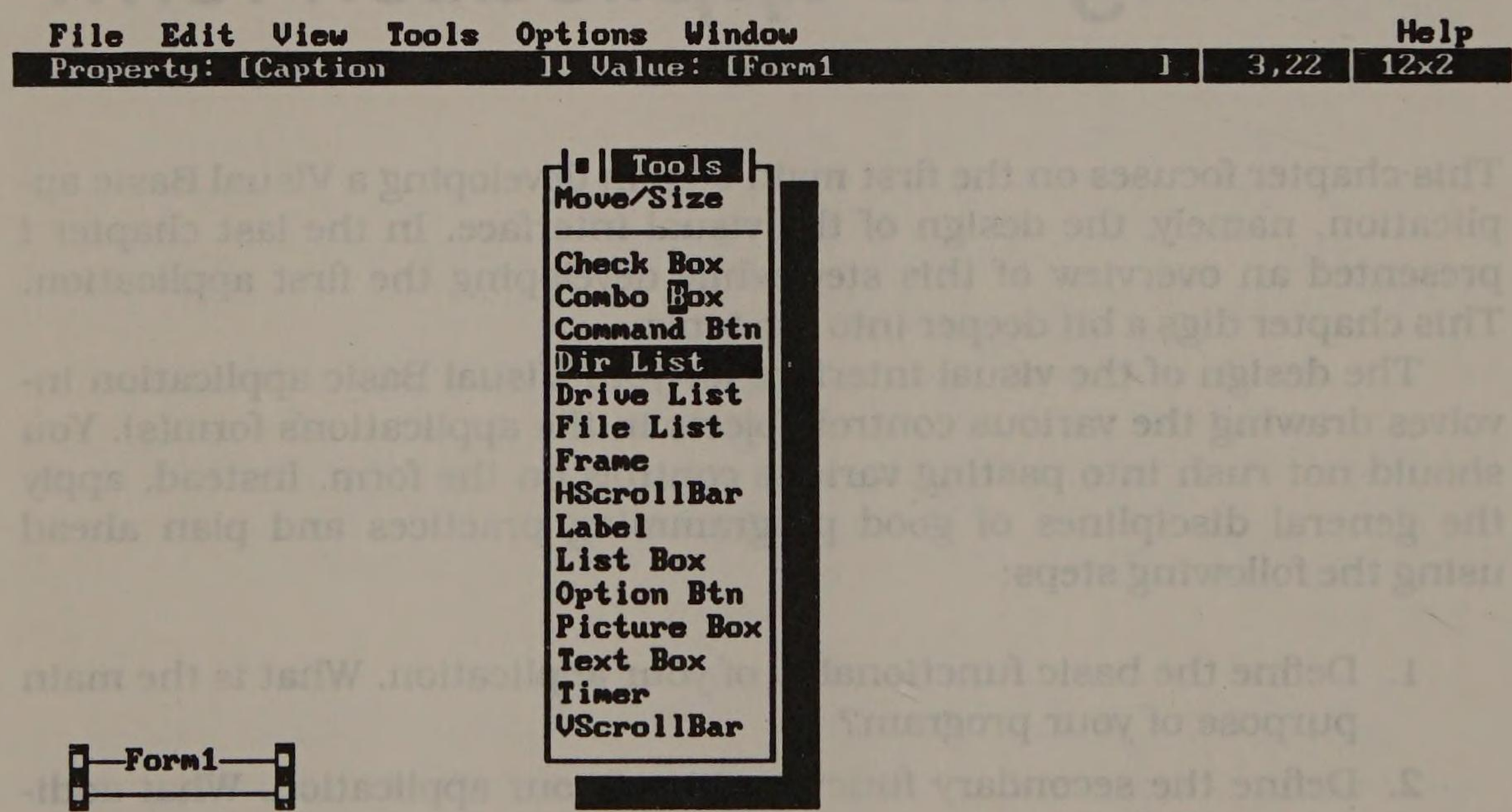
1. Define the basic functionality of your application. What is the main purpose of your program?
2. Define the secondary functionality of your application. What additional features do you want to implement?
3. Specify how you want your application to interact with the user.
4. Based on the above steps, sketch the interface on a piece of paper. Using the storyboard concept, go over what happens when you interact with the interface which you have just sketched. This is the first iteration of envisioning your application at work. Change the sketched interface as needed. You might specify "bells and whistles" features that make your Visual Basic program stand apart from others. Repeat this step as many times as needed.
5. Using the final version of your sketched interface, start Visual Basic and begin to draw the controls on the form. I recommend that you apply the storyboard concept one more time as you view the actual

controls in the form. This enables you to make any last minute changes.

These suggested steps indicate that care should be applied when designing the form of an application. It is easier to redesign on a piece of paper than on an Visual Basic form, especially if you have a complex application interface. In addition, the nature of Visual Basic applications does promote the use of the storyboard concept to visualize the animated application.

The previous recommended steps might seem trivial and like common sense advice. However, the easy visual programming style of Visual Basic will most likely lure you into jumping ahead and drawing controls—after all, drawing these controls is a fun process, especially when you are new to it.

Programming tip Plan ahead the interface of your Visual Basic application.



4-1 The toolbox.

The ToolBox

To make the discussion for creating the interface of a Visual Basic application a bit clearer, I now introduce you to the set of control objects available through the Toolbox. The Toolbox contains the lists of the various controls, as shown in FIG. 4-1. Here is a brief description for each control:

The picture box Displays text and extended ASCII characters emitted by the PRINT statement. The contents of a picture box cannot be modified. You can include other controls in a picture box.

The Text box Used in displaying text and accepting input from the user. The text box can display text in multiple lines when the MultiLine property is set to TRUE. The ScrollBars properties empower you to include vertical and/or horizontal scroll bars to the text box, enabling you to view a text from an ASCII file.

The Command button (Also known as the push button.) The control that you click on or push (normally by first selecting the button and then pressing Enter) to carry out a task.

The Option button (Also called the radio button.) An on/off button. Normally, you have a group of option buttons that provide you with a set of alternate choices, such that only one choice is made at any time (you can say that each option is antagonistic to all other options). When you turn on an option button, all of the other option buttons in the same group are turned off. This single selection resembles the button of a radio (whence the name radio button) that allows you to select only one radio station at any time.

The List box A control that shows a list of items from which you can select one. Items can be added or removed from the list at run time. If the size of the list box is insufficient to display all of the items in the list, a scroll bar is automatically added.

The Vertical scroll bar One of two controls (the other is the Horizontal scroll bar) that empowers you to quickly scroll through a long list of data. The vertical scroll bar can be used alone or in conjunction with a list box. When used alone, the scroll bar serves to select a numeric value for a setting (such as mouse sensitivity, keyboard speed, and graph scale).

The Drive list box Allows you to display the drives that are on-line and to select a current disk drive. This information is useful in determining the physical, RAM-based, and logical drives that are currently on-line in your system.

The File list box Enables you to locate, display, and select files in the current directory. The file list also allows you to open files for reading, writing, and random-access I/O. You can select the filenames that appear in the list by specifying file attributes, such as wildcard filenames.

The Label A control that displays fixed text on the form. I am using the term "fixed" here to stress that you cannot change the text of a label as easily as that of a text box. The label can be altered at run time from within a procedure.

The Frame control Serves to group a family of related controls (such as the option buttons or the check boxes). The grouping of controls occurs in two ways, logically and visually. The logical aspect is relevant to the functionality of the code. The visual aspect benefits the application end-user by introducing more visual organization to the form and adding clarity and ease of use.

The Check box A control that is similar to the option button. However, unlike the alternate choices offered by a group of option buttons, check boxes allow you to select multiple options. For example, check boxes can be used in searching for text. The typical check box options are: search forward, case-sensitive, and whole word. You can select any or all of these options. The combination of your choices defines the exact behavior of the related task.

The Combo box A control that blends or combines (whence the nickname Combo) the characteristics of a text box and a list box. A combo box allows you to make a selection by either typing it in or by choosing an item from the accompanying list box.

The Horizontal scroll bar A control that is similar to the vertical scroll bar. The difference between the two types of scroll bars is their visual orientation.

The Timer A special control. As the name suggests, the timer runs code at preset regular time intervals. Timer controls are visible at design time, but not at run time.

The Directory list box A control that displays the directory tree hierarchy and allows you to select a new directory at run time.

The drive, directory, and file list boxes work together in viewing and accessing files anywhere in your system. Each list box contributes to the search for the sought files.

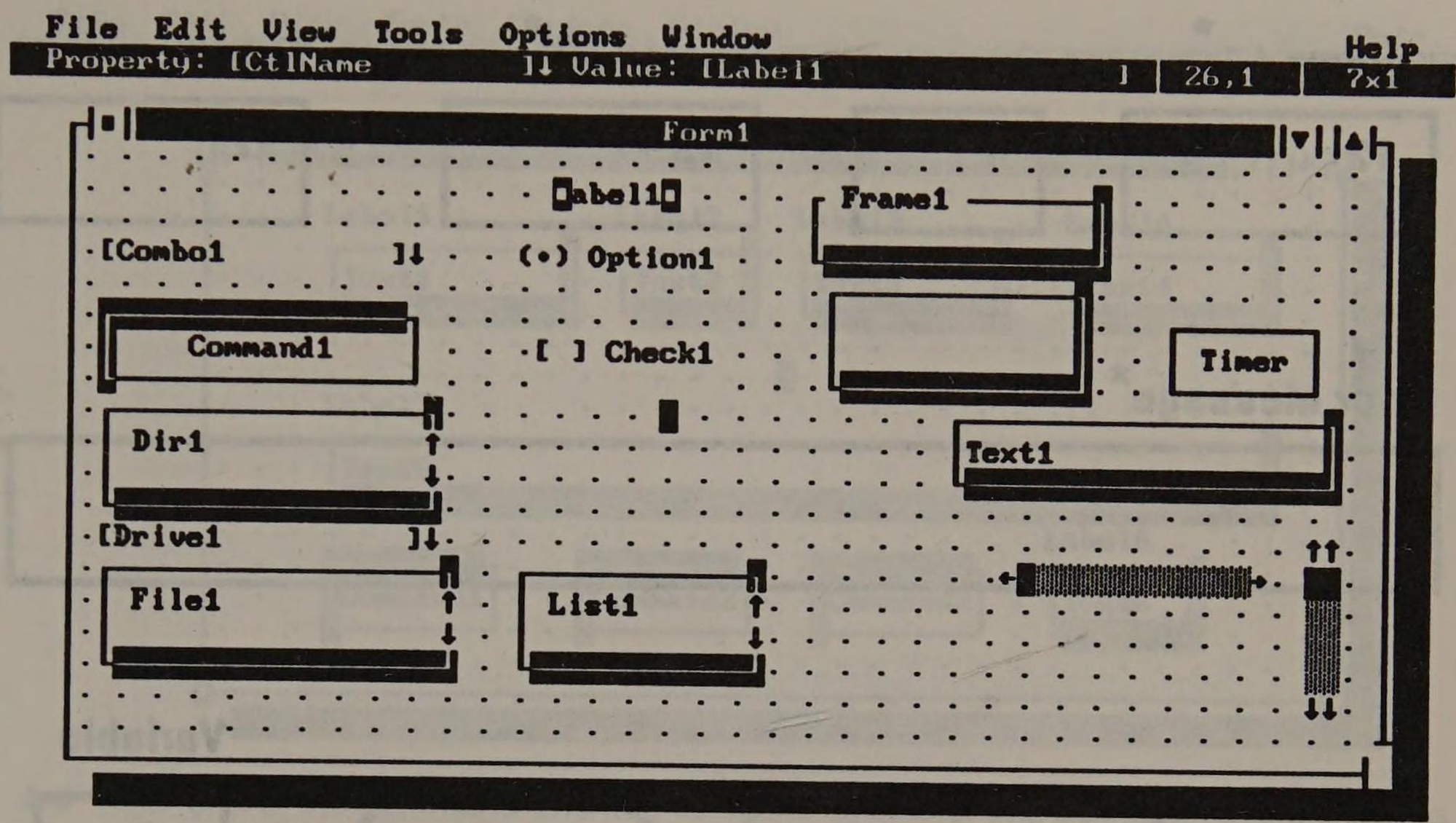
Figure 4-2 shows a form with each of the above controls (with default settings).

✍ **Programming note** Each control has a predefined set of properties, events it can handle, and methods. The properties fine-tune the visual appearance and behavior of the control object. The predefined events determine the type of response that can be obtained from the control. The preset methods are special preprogrammed procedures that are used to manipulate the control at run time. The language reference manual describes in detail the properties, events, and methods for the various controls.

The command-oriented scientific calculator

Let's look at drawing the interface of a new Visual Basic application. Following the steps that I described earlier, I begin by defining the various levels of functionality for the new command-oriented scientific calculator (COSC). I am using the term command-oriented to indicate that the interface uses a few text boxes to store the operands, the operator or math function, and the result. The advantage of this kind of calculator is that you can add more functions and operators without changing the interface.

The secondary level functionality includes the following features:



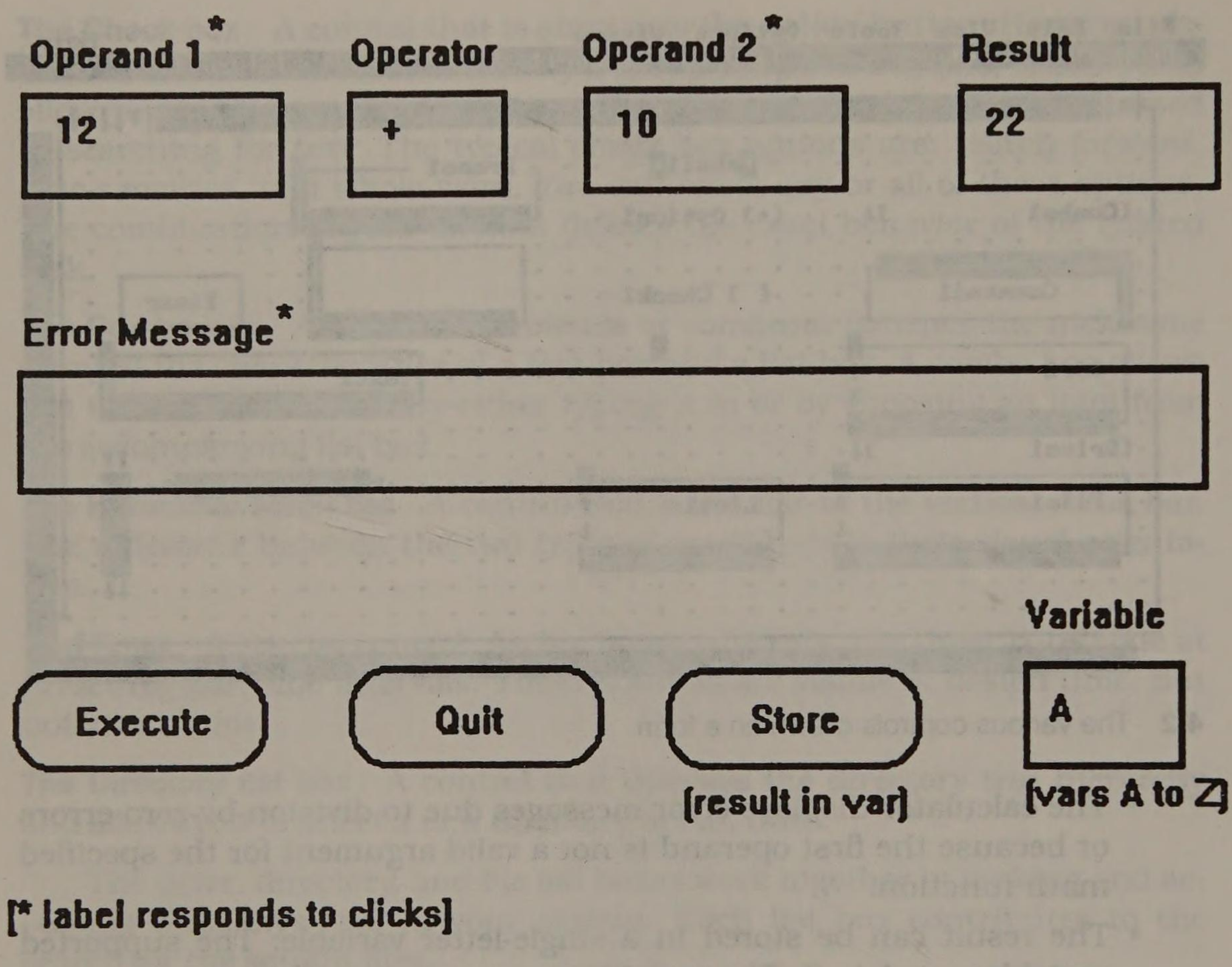
4-2 The various controls drawn on a form.

- The calculator displays error messages due to division-by-zero errors or because the first operand is not a valid argument for the specified math function.
- The result can be stored in a single-letter variable. The supported variables are A to Z. The variable names are not case-sensitive.
- The operands may be either a numeric constant or a variable.

The third step in creating the interface involves specifying how the application interacts with the user:

- The user keys in the operands and operator (or math function) in their respective text boxes.
- Either the specified operation or math function is executed by pressing an execute command button.
- The result of a valid math operation or function evaluation is placed in the result text box.
- The name of a variable used to store the current result is typed in a special text box.
- A store command button allows the user to store the result in the specified variable.
- All error messages are displayed in a dedicated text box.
- A quit command button enables the user to exit the application.

The fourth step involves sketching the interface of the application.

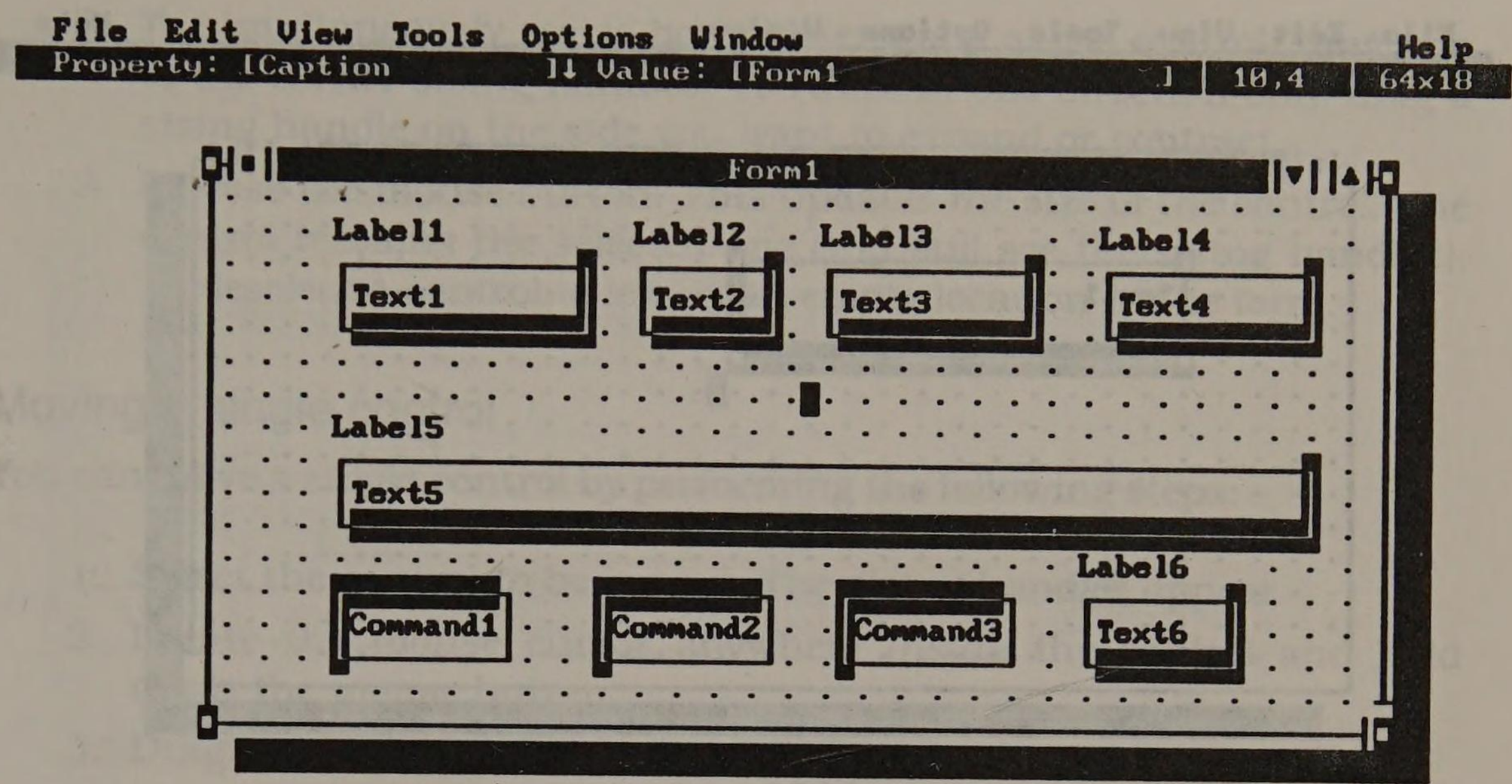


4-3 The PaintBrush sketch for the command-oriented scientific calculator.

Figure 4-3 shows such a sketch, that was drawn using PaintBrush. The sketch includes a few comments that I have enclosed in square brackets. I have also marked a few labels with the asterisk symbol to indicate that these labels respond to a click. This addition represents the part of adding the "bells and whistles" to the application. To substitute the name of a variable in either the operand box with the value store in that variable, click on the corresponding label. To clear the contents of the error message box, click on the associated label.

Drawing the controls

Armed with a stable design of the interface, let's start creating the Visual Basic form for the COSC application. I will carry out the form design process in two phases. In the first phase, presented in this chapter, the required controls are drawn. In the second phase, presented in the next chapter, I will fine tune the settings of the various controls. The aim of the first phase is to create a form with controls that resemble FIG. 4-4.



4-4 The form of the command-oriented scientific calculator (COSC) application, showing the default settings.

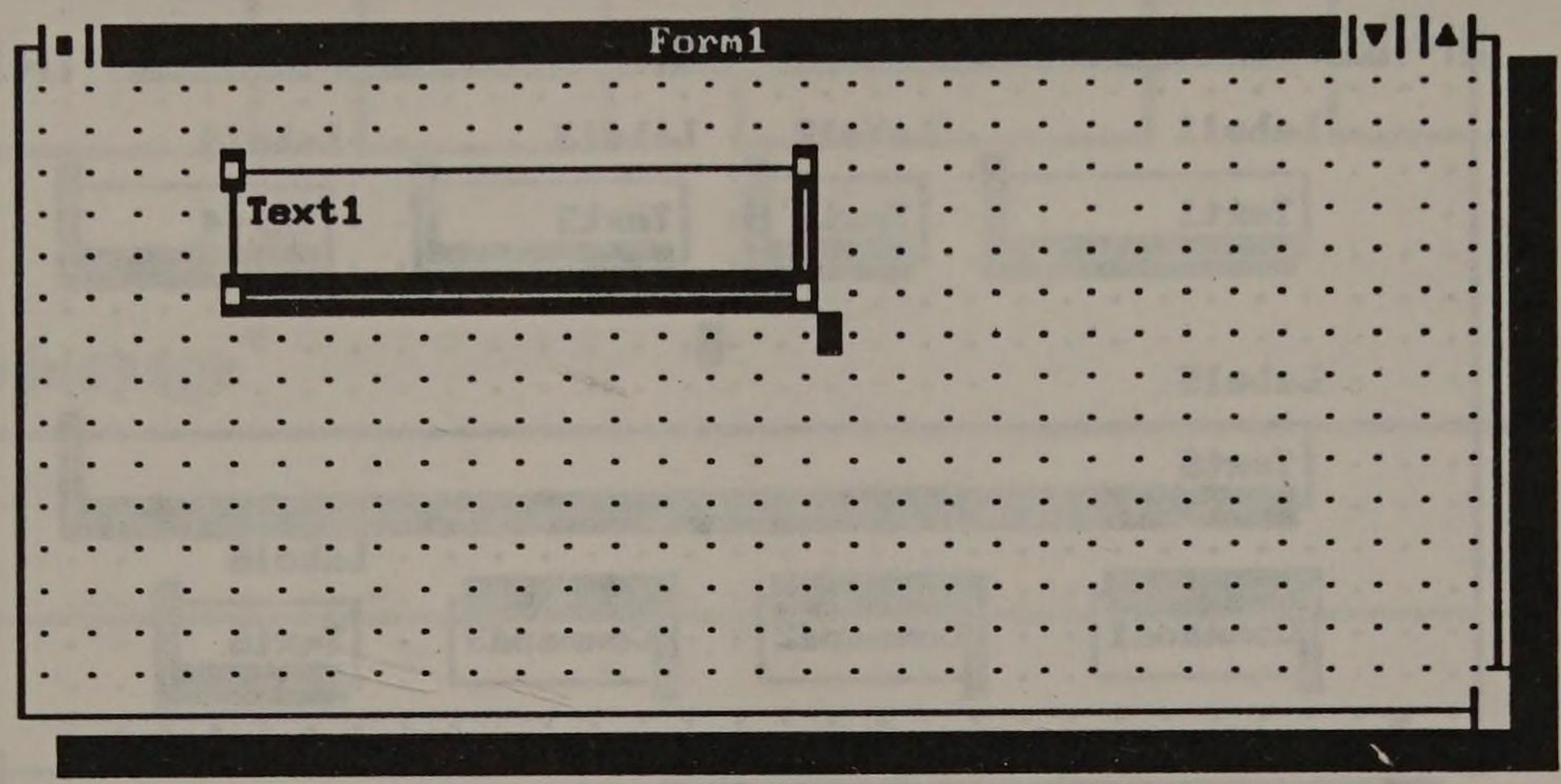
Drawing a control

There are two methods to draw a control: the normal method and the shortcut method. For example, to draw the first text box using the normal method, apply these steps:

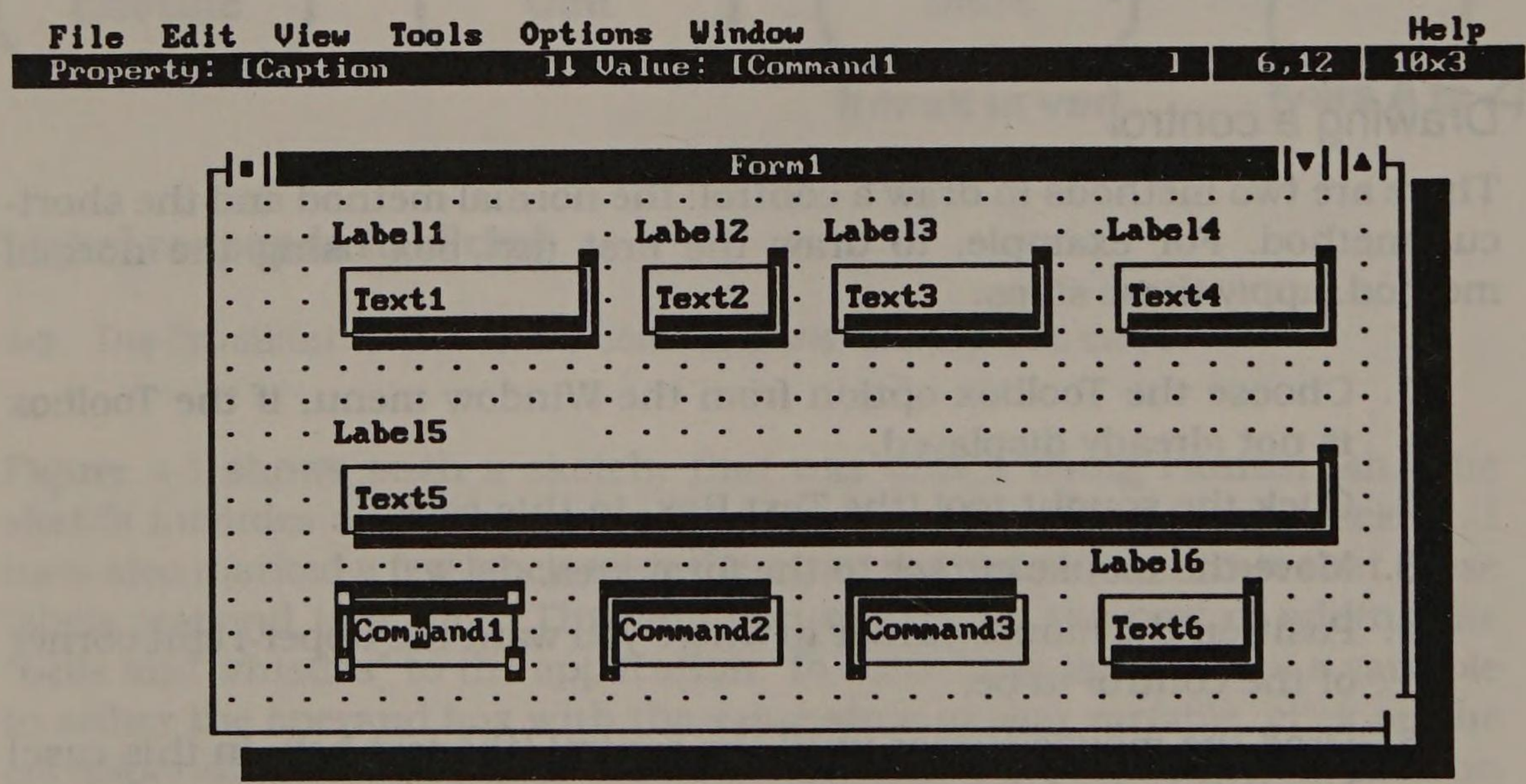
1. Choose the Toolbox option from the Window menu, if the Toolbox is not already displayed.
2. Click the sought tool (the Text Box, in this case).
3. Move the mouse cursor to the form area.
4. Position the mouse cursor to where you want the upper-right corner of the control to be.
5. Drag the mouse cursor until the control (the text box, in this case) has the desired dimensions.
6. Release the mouse button. The control appears (in this case, a text box with the string "Text1" appears) on the form and the arrow-shaped mouse cursor is restored. Figure 4-5 shows a snapshot taken while drawing the first text box.

Repeat the above process for the other controls shown in FIG. 4-4.

☞ The shortcut method works by double clicking the sought control in the Toolbox. This causes the chosen control to appear in the middle of the form. You then need to move, and most likely resize, the control.



4-5 Drawing the first text box of the COSC application.



4-6 A selected command button.

Resizing a control

To resize a control, perform the following steps:

1. Select the control to be resized by clicking on it. This causes a number of small black rectangles, called the sizing handles, to appear at the edge of the control object. Figure 4-6 shows a selected command button.

2. To simultaneously resize both dimensions of the control, drag one of the corner sizing handles. To resize in one direction only, drag a sizing handle on the side you want to expand or contract.
3. Release the mouse button. This updates the size of the control. The control remains the selected one (you still see the sizing handles). To deselect a control, click on an empty location in the form.

Moving a single control

You can move a single control by performing the following steps:

1. Select the control to be moved. The sizing handles appear.
2. Locate the mouse cursor anywhere *inside* the control and hold down the mouse button.
3. Drag the control to its new position on the form.
4. Release the mouse button.

Moving multiple controls

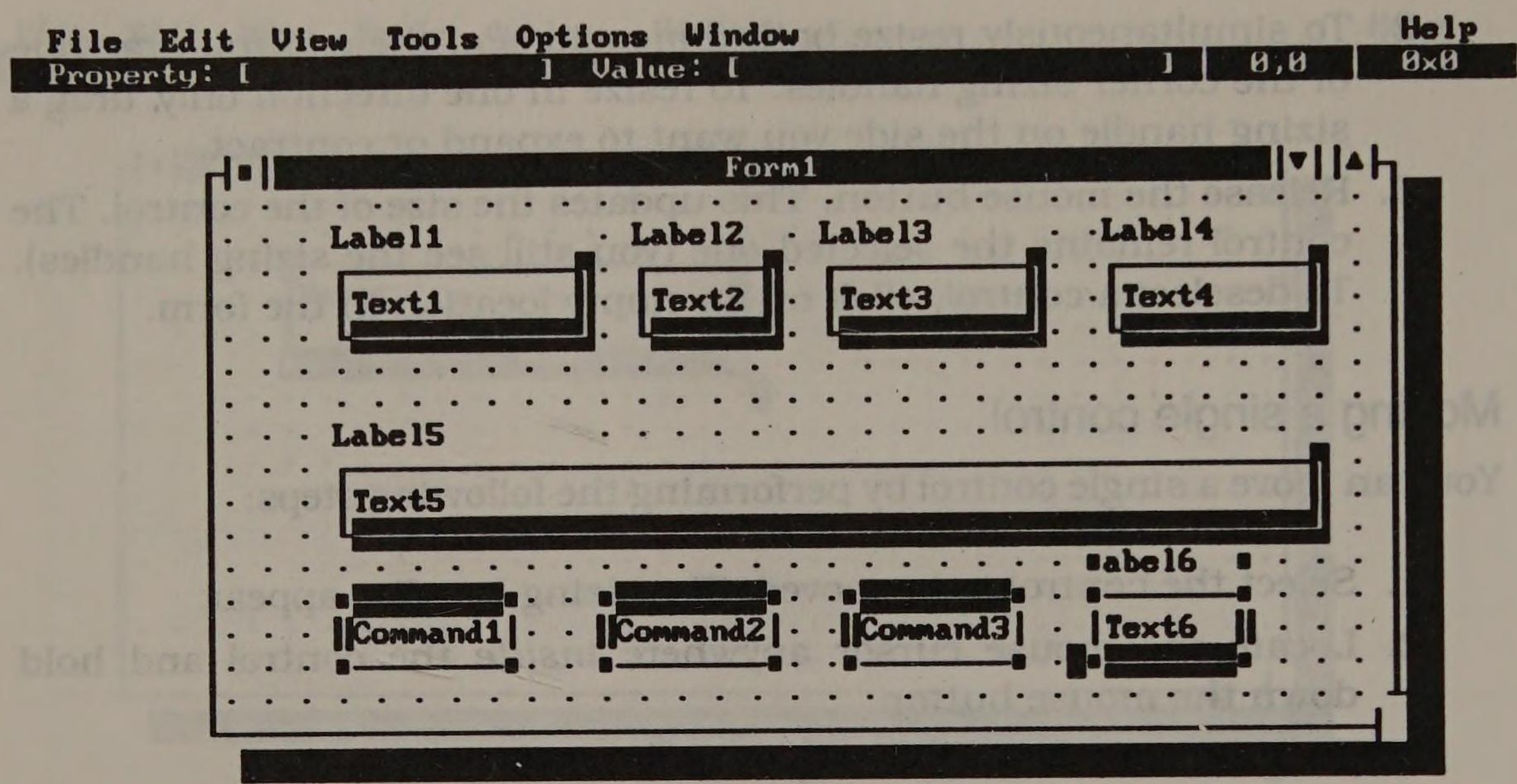
You also can move a group of controls at one time. This causes the controls to be displaced by the same distance and direction. You can move a group of controls by carrying out the following steps:

1. Select the first control by clicking on it.
2. To select all subsequent controls, hold the CTRL key and click on each of the other target controls. Notice first, the sizing handles change color from black to gray. Second, the sizing handles appear on all of the selected controls. To deselect any control, click on that control while holding down the CTRL key. In effect, what you are doing is toggling the selection of that control. Figure 4-7 shows a group of selected controls in the form of the COSC application.
3. Drag the set of selected controls to a new position on the form.
4. Release the button to stop moving the controls. Click on a clear form area to deselect all of the controls.

Deleting a single control

You can delete a single control by performing the following tasks:

1. Select the control you want to delete by clicking on it.
2. Press the DEL key (some keyboards spell out delete). You can also remove a control by selecting the Delete option from the Edit menu.



4-7 A group of selected controls in the form of the COSC application.

Deleting multiple controls

You can delete multiple controls in one swoop. Follow these steps:

1. Select the first control you want to delete by clicking on it.
2. To select all subsequent controls, hold the CTRL key and click on each of the other target controls. To deselect any control, click on that control while holding down the CTRL key.
3. Press the DEL key. You also can remove the group of controls by selecting the Delete option from the Edit menu.

Aligning the controls

The Visual Basic form uses a grid that determines the jumps made by a control when you move it. The grid is also very helpful in aligning controls. The grid spacing has the width and height of a character. You can toggle the visibility of the grid by using the Grid Lines option in the View menu.

Saving your work

Save the form and project using the SCICALC.FRM and SCICALC.MAK filenames, respectively. The next chapter looks at altering the properties of the controls in more detail.

Documenting the Visual Basic forms

The new visual programming approach of Visual Basic brings with it new requirements for documenting your applications. This becomes an even

more formidable challenge because the Visual Basic code is not linear (as is the case with QuickBasic or GW-Basic). The nonlinear nature of the Visual Basic code comes from the fact that code is dispersed—it is attached to the various forms and control objects. Therefore, it is a good practice to document the various components of an application.

Application Name: _____
Application Code Name: _____
Version: _____ Date Created: _____

Programmer(s): _____
Company: _____

List of Filenames

Storage Path: _____
Project _____
Global _____

Form 1 _____	Module 1 _____
Form 2 _____	Module 2 _____
Form 3 _____	Module 3 _____
Form 4 _____	Module 4 _____
Form 5 _____	Module 5 _____
Form 6 _____	Module 6 _____
Form 7 _____	Module 7 _____
Form 8 _____	Module 8 _____
Form 9 _____	Module 9 _____

4-8 A sample for the basic design documentation.

Figure 4-8 shows a sample for the basic design documentation. The suggested documentation form lists the application name, code name, version number, date of creation, programmers, company, and the list of application files—the project file, the global module file, the form file(s), and the module file(s). The global module file contains mainly global constants and variables. As far as Visual Basic is concerned, the global module is just another module. The global status of that module is merely conceptual.

Figure 4-9 presents a documentation form for the individual Visual Basic form. The information includes the form number, form filename, version number, date of creation, and a list of control objects. This list tabulates the type of control, the default control name (as initially assigned by Visual Basic), the purpose of the control, the location of its upper-right corner, and the dimensions of the control (length and width). The last two attributes are optional—they are needed only when the location and size of the control are important. The next two chapters present other documentation forms that keep track of the custom settings.

Applying the documentation forms of FIGS. 4-8 and 4-9 to the COSC ap-

Form # _____ Form Filename: _____
Version: _____ Date: _____

Control object type	Default	CtlName	Purpose
(location) (dimensions)			

Note: The "location" refers to the coordinates of the upper right corner of the object. The "dimensions" refer to the length and width of the objects. All dimensions are expressed in characters.

4-9 The documentation for the individual.

plication yields FIGS. 4-10 and 4-11. Notice that FIG. 4-11 does not include the location and size of the controls, because such information is not critical to the application's interface. In other words, you might eyeball the location and size of the controls when you try to reproduce the COSC interface yourself. Had the location and size of the controls been vital, FIG. 4-11 would have surely included them—and you would have been required (for whatever reason) to draw the controls according to the supplied data.

Application Name: Command-oriented Scientific Calculator
Application Code Name: COSC
Version: 1.0 Date Created: June 18, 1992
Programmer(s): Namir C. Shammas
Company: N/A

List of Filenames

Storage Path: C:\VBDOS\EASYVB
Project SCICALC.MAK
Form 1 SCICALC.FRM

4-10 A sample for the basic design documentation.

Form # 1
Version: 1.0

Form Filename: SCICALC.FRM
Date: June 18, 1991

Control object type	Default CtlName	Purpose
Text Box	Text1	Key in operand 1.
	Text2	Key in operator.
	Text3	Key in operand 2.
	Text4	Display the result.
	Text5	Display error message.
	Text6	Key in the variable names.
Label	Label1	Label of Text1. Replace variable with numeric value.
	Label2	Label of Text2.
	Label3	Label of Text3. Replace variable with numeric value.
	Label4	Label of Text4.
	Label5	Label of Text5. Clear error message.
	Label6	Label of Text6.
Command button	Command1	Execute math operation.
	Command2	Quit.
	Command3	Store the result in a variable.

4-11 The documentation for the individual controls of SCICALC.FRM.

5

Changing control values

The predefined visual controls that you draw in a form come with a built-in set of properties. These properties determine the visual appearance and the behavior of the various objects (form and controls) especially during program execution. Each property has a default value which is provided by Visual Basic. While these "factory-supplied" defaults are suitable for most applications, Visual Basic allows you to alter them to fine-tune your application.

Table 5-1 lists the properties of the various Visual Basic objects. The table associates each control with a data type and lists the objects that own that property. This table should give you a clear idea of the important role of properties in Visual Basic. The *Microsoft Visual Basic Language Reference* manual describes each property. Keep in mind that the same property might play a somewhat different role in the various objects that own that property.

Working with the Properties bar

The Properties bar contains two combo boxes that enable you to view and edit the values for the forms and control objects (some values are not available during design time). The Property combo box contains an ordered list of properties. The Value combo box contains the current values. Let's first see how to select properties to view their values.

Viewing properties and values

The first step in viewing the properties and values of an object involves selecting that object. If you need to select the form, click anywhere on the clear area of the form. To select a control object, click on that object. The Proper-

Table 5-1. The Visual Basic properties.

Property	Data type	(Access mode) Applies to
ActiveControl*		(O) Screen
ActiveForm*		(O) Screen
Alignment	integer 0 to 2	(R) Label
Archive	TRUE/FALSE	(R) File
Attached	TRUE/FALSE	(O) Scroll
AutoRedraw	TRUE/FALSE	(R) Form, Picture
AutoSize	TRUE/FALSE	(R) Label, Picture
BackColor	long integer	(R) Check, Clipboard, Command, Dir, Drive, File, Form, Frame, Label, List, Option, Picture, Printer, Text
BorderStyle	integer	(O) Form, Picture, Text (R) Label
Cancel	TRUE/FALSE	(R) Command
Caption	string	(R) Check, Command, Form, Frame, Label, Menu, Option
Checked	TRUE/FALSE	(R) Menu
ControlBox	TRUE/FALSE	(O) Form
ControlPanel	integer	Screen
CtlName	string	(X) Check, Combo, Command, Dir, Drive, File, Frame, Label, List, Menu, Option, Picture, Scroll, Text, Timer
CurrentX*	integer	(R) Form, Picture, Printer
CurrentY*	integer	(R) Form, Picture, Printer
Default	TRUE/FALSE	(R) Command
DragMode	integer 0 to 1	(R) Check, Combo, Command, Dir, Drive, File, Frame, Label, List, Option, Picture, Scroll, Text
Drive*	string	(R) Drive
Enabled	TRUE/FALSE	(R) Check, Combo, Command, Dir, Drive, File, Form, Frame, Label, List, Menu, Option, Picture, Scroll, Text, Timer
Filename*	string	(R) File
ForeColor	long integer	(R) Check, Combo, Dir, Drive, File, Form, Frame, Label, List, Option, Picture, Printer, Text
FormName	string	(X) Form
Height	single	(R) Check, Combo, Command, Dir Drive, File, Form, Frame, Label, List, Option, Picture, Printer, Screen, Scroll, Text
Hidden	TRUE/FALSE	(R) File
Index	integer	(O) Check, Combo, Command, Dir

		Drive, File, Frame, Label, List, Menu, Option, Picture, Scroll, Text, Timer
Interval	integer	(R) Timer
LargeChange	integer	(R) Scroll
Left	single	(R) Check, Combo, Command, Dir Drive, File, Form, Frame, Label, List, Option, Picture, Scroll, Text
List*	string	(R) Combo, List (O) Dir, Drive, File
ListCount*	integer	(O) Combo, Dir, Drive, File, List
ListIndex*	integer	(R) Combo, Dir, Drive, File, List
Max	integer	(R) Scroll
MaxButton	TRUE/FALSE	(O) Form
Min	integer	(R) Scroll
MinButton	TRUE/FALSE	(O) Form
MousePointer	integer 0 to 12	(R) Check, Combo, Command, Dir Drive, File, Form, Frame, Label, List, Option, Picture, Screen, Scroll, Text
MultiLine	TRUE/FALSE	(R) Text
Normal	TRUE/FALSE	(R) File
Parent*	form	(O) Check, Combo, Command, Dir, Drive, File Frame, Label, List, Menu, Option, Picture, Scroll, Text, Timer
Path*	string	(R) Dir, File
Pattern	string	(R) File
PrintTarget	string	Printer
ReadOnly	TRUE/FALSE	(R) File
ScaleHeight	single	(R) Form, Picture, Printer
ScaleWidth	single	(R) Form, Picture, Printer
ScrollBars	integer 0 to 3	(O) Text
SelLength*	long integer	(R) Combo, Text
SelStart*	long integer	(R) Combo, Text
SelText*	string	(R) Combo, Text
SmallChange	integer	(R) Scroll
Sorted	TRUE/FALSE	(O) Combo, List
Style	integer 0 to 2	(O) Combo
System	TRUE/FALSE	(R) File
TabIndex	integer	(R) Check, Combo, Command, Dir Drive, File, Frame, Label, List, Option, Picture, Scroll, Text
TabStop	TRUE/FALSE	(R) Check, Combo, Command, Dir Drive, File, List, Option, Picture, Scroll, Text

Table 5-1 Continued.

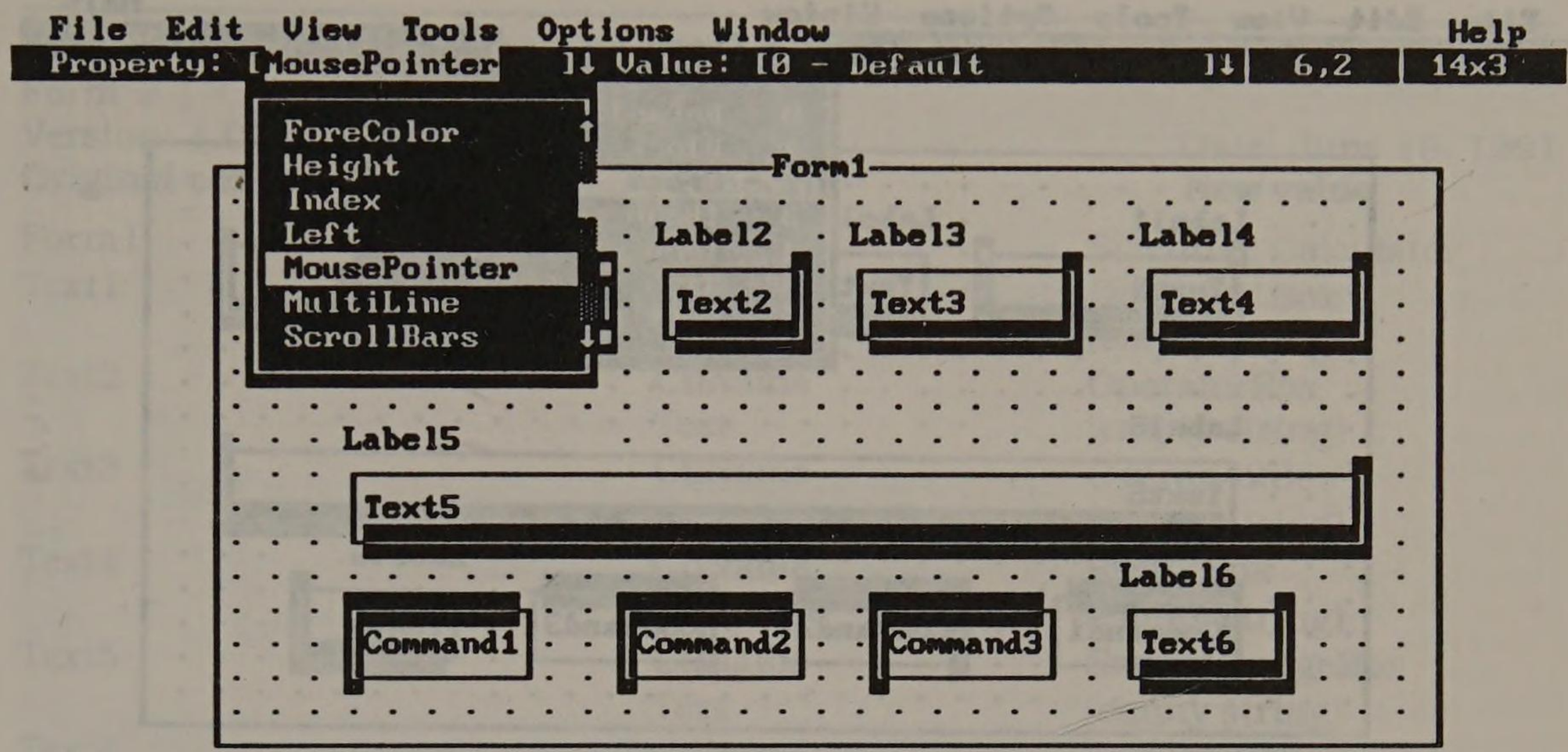
Tag	string	(R) Check, Combo, Command, Dir Drive, File, Frame, Label, List, Menu, Option, Picture, Scroll, Text, Timer
Text	string	(O) List* (R) Combo, Text
Top	single	(R) Check, Combo, Command, Dir Drive, File, Form, Frame, Label, List, Option, Picture, Scroll, Text
Value	integer	(R) Check, Command, Option, Scroll
Visible	TRUE/FALSE	(R) Check, Combo, Command, Dir, Drive, File, Form, Frame, Label, List, Menu, Option, Picture, Scroll, Text
WindowState	integer 0 to 2	(R) Form

Access mode code is:
R read/write at run time.
O read-only at run time.
X not available at run time.

The * symbol indicates that the property (or properties) is not available at design time.

TRUE/FALSE indicates a sepcial enumerated integer, such that:
TRUE = −1
FALSE = 0

ties bar displays the current property and its value for the currently selected item. It is interesting to point out that if you select multiple controls, the combo boxes of the Properties bar go blank. The Property combo box displays the currently selected property. To select another property click on the downward-pointing arrow in the combo box. This causes the scrollable list of properties to appear. You can scroll through the list either using the mouse or the cursor keys. The current entry is displayed as selected text (in white characters with a black background). You can jump to certain entries by typing the first letter of that entry. For example, if you are inspecting the properties of a text box, you can zoom to the Text property by typing the letter T (either uppercase or lowercase) for a few times. Every time you press T the combo box selects the next item that starts with that letter. Press Enter to select the current entry from the list. Figure 5-1 shows the Property combo box with the list of properties. When you select a new property, the corresponding value is displayed in the value combo box.



5-1 The Properties bar and the Property combo box.

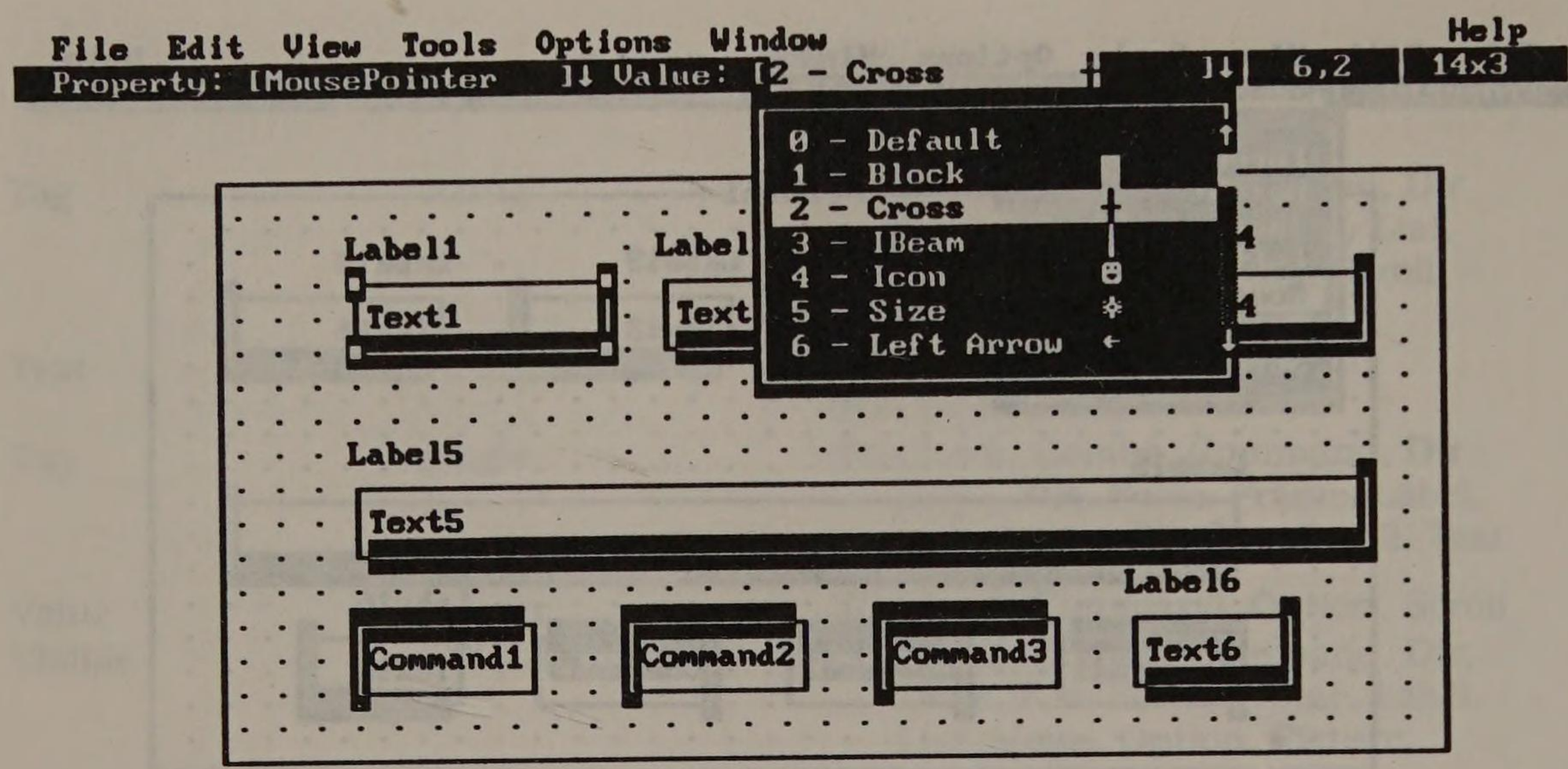
Changing the current value

The first steps involved in changing the current value of an object are the same ones for viewing the object's values. Because there are two basic types of values, there are also two basic methods for changing these values. The first type of value contains discrete (also called enumerated) integer values. The most popular example is the True (−1) and False (0) values.

When you select the value combo box, notice if the accompanying arrow-button is active or not. If the button is active, then you are dealing with an enumerated value. In this case, click on the arrow button to reveal the available choices. You can select a new value by scrolling through the list of values and then pressing Enter or by clicking on the single check button located to the left of the Value combo box.

You also can zoom in on an enumerated value by typing the first letter of that value. For example, you can select False from a True/False enumerated list by simply pressing the F key. To reject the new selection and resort to the previous one, click the X button located just to the left of the single check button. Figure 5-2 shows the value combo box displaying an enumerated True/False choice.

The second kind of values contains data of the various basic types, such as integers, long integers, strings, and singles. You can type any value that is within the valid range for the corresponding data type of that value. To enter a new value, simply type in the new value. Press Enter or click on the single check button to accept the new value. Click on the X button to resort to the previous value.



5-2 The Properties bar and the Value combo box.

Application (Code) Name: _____

Form # _____

Version: _____ Date: _____

Original control name	**Property**	**New value**

5-3 The suggested form for documenting changes in the form and control values.

Documenting custom values

It is a sound programming practice to keep track of the changes you make to the values of the form or controls. Documenting these changes permits you (or anyone else) to easily keep track of application's objects. In the last chapter I presented two documentation forms that can be used to describe the initial phases of building a Visual Basic application.

Now I present FIG. 5-3, which contains a suggested documentation form for keeping track of the value changes. As the number of such forms increases, we need to establish a link between the information in these forms. Borrowing from the basic notions of relational databases, I recommend the use of the default form and control names as a way to keep track of the various objects.

Ironically though, you invariably change these control names to better describe the role of the control in your application. Nevertheless, I feel that the default object names can serve us well in tracing the application's objects from the moment you create them. Using the default control name, you are able to tie in the formation in FIG. 5-3 with that of FIG. 4.10—relating the control type, purpose, and new value.

Figure 5-4 shows the suggested documentation form with the custom values data for the COSC application. The data for these values are used in

Application (Code) Name: COSC
Form # 1
Version: 1.0
Date: June 19, 1991

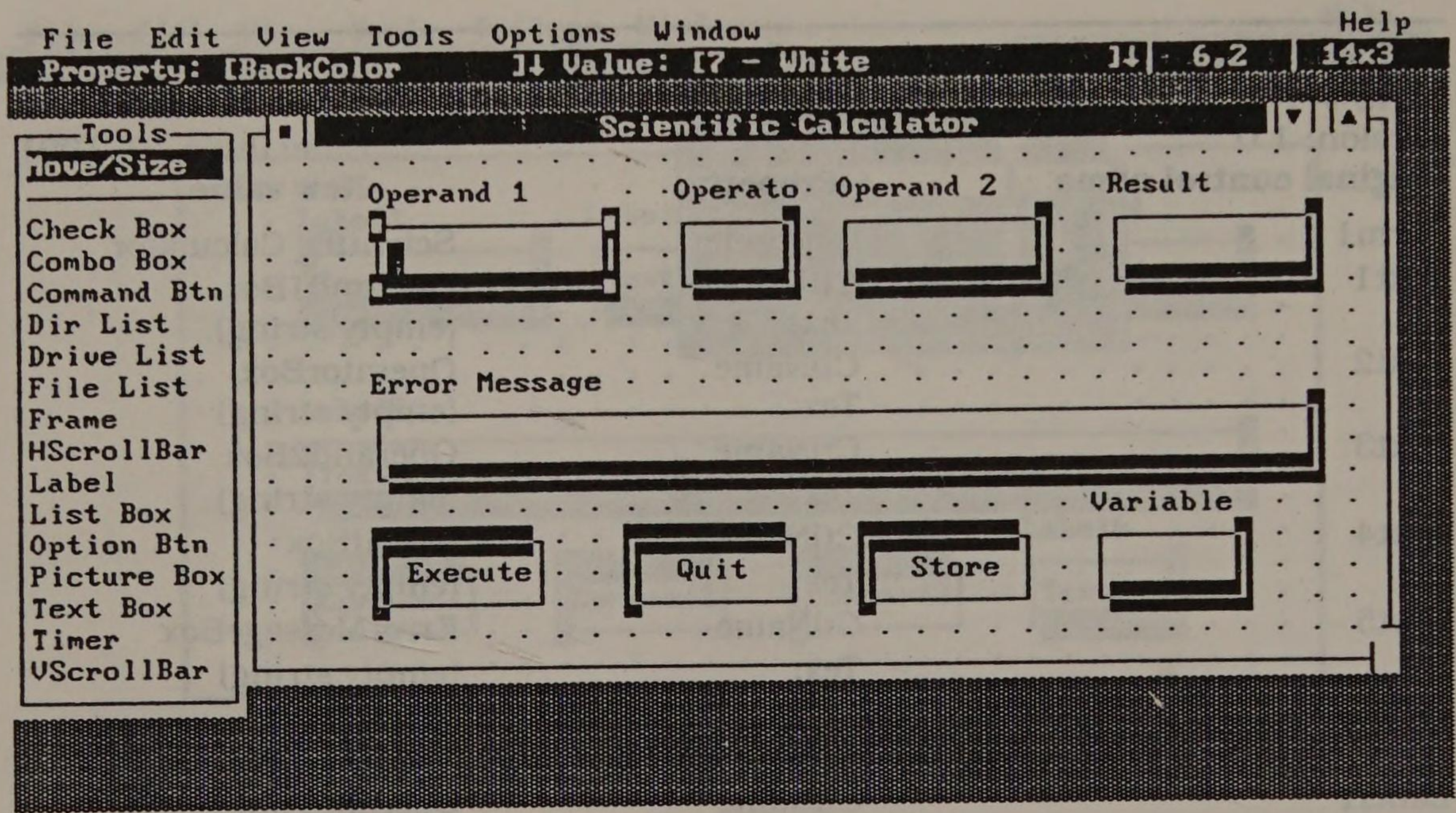
Original control name	Property	New value
Form1	Caption	Scientific Calculator
Text1	CtlName	Operand1Box
	Text	(empty string)
Text2	CtlName	OperatorBox
	Text	(empty string)
Text3	CtlName	Operand2Box
	Text	(empty string)
Text4	CtlName	ResultBox
	Text	(empty string)
Text5	CtlName	ErrorMessageBox
	Text	(empty string)
Text6	CtlName	VariableBox
	Text	(empty string)
Label1	CtlName	Operand1Lbl
	Caption	"Operand1"
Label2	CtlName	OperatorLbl
	Caption	Operator
Label3	CtlName	Operator2Lbl
	Caption	Operand2
Label4	CtlName	ResultLbl
	Caption	Result
Label5	CtlName	ErrorMessageLbl
	Caption	Error Message
Label6	CtlName	VariableLbl
	Caption	Variable
Command1	CtlName	ExecuteBtn
	Caption	Execute
Command2	CtlName	QuitBtn
	Caption	Quit
Command2	CtlName	StoreBtn
	Caption	Store

5-4 The changes in the values for the COSC application.

the next section when I describe how to alter the values for form and controls.

Changing the values for the scientific calculator

Using the instructions described earlier, change the values listed in FIG. 5-4 to obtain the form shown in FIG. 5-5. You might need to load the COSC project again by opening the SCICALC.MAK project (in the \VBDOS\EASYVB directory). Open the form by double clicking on the form file. To alter the values of the Text1 text box, for example, perform the following steps:



5-5 The modified form for the COSC application.

1. Select the Text1 text box by clicking on it.
2. Select the Property combo box. If the current property is not CtlName, scroll through the list until you find it. When the CtlName property is highlighted, press the Enter key to select it.
3. The corresponding value is Text1, the default value supplied by Visual Basic. Click on the Value combo box and type Operand1Box. Press the Enter key when you have finished.
4. Select the Property combo box again. Scroll through the list (or press the key T) until the Text property becomes highlighted. Press Enter to select the Text property.
5. The corresponding value is Text1, the default contents of the selected text box. Click on the Value combo box and press the space bar, followed by the Backspace key. This makes the current value a null string. Press the Enter key to accept the new value.

You can perform the other changes in a manner similar to the way you changed the values of the Text1 text box—change all of the values of the same control before moving on to the next control. There is another approach that is a bit easier and faster. It involves the following steps:

1. Change the values of similar controls, one category at a time. For example, start by editing all the values of the text box, then the command buttons, and then the labels.
2. While working with each group of similar controls, edit the value of the same property before moving on to the next property. When you

edit a property of a control, that property remains the selected one when you select the next similar control. This feature reduces the number of searches you need to perform in the Property combo box. In addition, you can focus on entering or selecting the new value.

When you have finished changing the values, save the form and project files, using the Save File and Save Project options in the File menu.

6
Attaching code to the controls

This chapter looks at the third general step involved in developing a Visual Basic application. In chapter 4, the form and controls were drawn for the COSC application. In chapter 5, the settings for the form and the controls were adjusted. In this chapter we attach the code to the various controls to bring the application to life.

Coding a Visual Basic application

Visual Basic is an easy-to-use program development tool. The reason for this is the visual programming approach employed by Visual Basic. Rather than starting from scratch, you are provided with a well-developed set of visual objects. A lot of programming has been done by the designers of Visual Basic to provide the various objects their visual appearances and behaviors. This is translated into the following three major areas: properties, events, and methods. The predefined properties can be regarded as parameters or data fields that set or query the appearance and behavior of the various Visual Basic objects. The predefined set of events determines how an object may interact with the application user. Methods are special procedures that manipulate the various objects.

This chapter looks at coding the controls to respond to specific events. While each kind of control is able to respond to a fixed set of events, you rarely (at least for now) program any control to respond to that entire set of events. Typically, a control is programmed to respond to a few events. In the case of the COSC application, controls respond to one event—the mouse click. Because clicking on a control is common in Windows-like DOS applications, the Click event is among the most popular events handled.

In chapter 4, I began presenting the COSC application by discussing

the functionality of the program. Here is a recap of what the command-oriented scientific calculator basically does:

- The user types in the operands and the operator in their respective text boxes. Alternately, the user may type the argument of a function in the text box labeled "Operand 1" and enter the name of the math function in the Operator text box. The name of the function is not case-sensitive.
- The user clicks the Execute button to perform the desired operation or function evaluation. The result is calculated and displayed in the Result text box.
- The result can be stored in a single-letter variable (A to Z). This is carried out by typing the name of the target variable in the Variable text box and then clicking the Store button.
- Clicking on the "Operand 1" and "Operand 2" labels translates the name of a variable (if found) in the Operand 1 and Operand 2 text boxes.
- Clicking on the "Error Message" label clears the content of the Error Message text box.
- Clicking on the Quit key terminates the program.

Writing event-handling procedures

Every control that you draw on the form, as well as the form itself, can respond to a specific set of events. The number and types of events vary for each kind of control. Table 6-1 lists the Visual Basic events and the objects that respond to each type of event. In coding a Visual Basic application you must associate the different controls (the number varies between one control to all of the controls on the form if you have a very sophisticated application) to the events they should handle. The association between a control and an event is called code attachment. The procedure that implements an attached code derives its name from the name of the control and the name of the event it handles, using the following general syntax:

```
SUB objectName_eventName(⟨list of parameters⟩)
    sequence of Basic statements
END SUB
```

The object name for a control is taken from the setting of the CtlName property. In the case of the form itself, the object name is systematically taken as Form:

```
SUB Form_eventName(⟨list of parameters⟩)
    sequence of Basic statements
END SUB
```

The event-handling procedures resemble normal QuickBasic SUB procedures that you have most likely written as a programmer. The difference is

Table 6-1 The Visual Basic events

Event	Applies to
Change	Combo, Dir, Drive, Label, Picture, Scroll, Text, Timer
Click	Check, Combo, Command, Dir, File, Form, Label, List, Menu, Option, Picture
DblClick	Combo, File, Form, Label, List, Option, Picture
DragDrop	Check, Combo, Command, Dir, Drive, File, Form, Frame, Label, List, Option, Picture
DragOver	Check, Combo, Command, Dir, Drive, File, Form, Frame, Label, List, Option, Picture
DropDown	Combo
GotFocus	Check, Combo, Command, Dir, Drive, File, Form, List, Option, Picture, Scroll, Text
KeyDown	Check, Combo, Command, Dir, Drive, File, Form, List, Option, Picture, Scroll, Text
KeyPress	Check, Combo, Command, Dir, Drive, File, Form, List, Option, Picture, Scroll, Text
KeyUp	Check, Combo, Command, Dir, Drive, File, Form, List, Option, Picture, Scroll, Text
Load	Form
LostFocus	Check, Combo, Command, Dir, Drive, File, Form, List, Option, Picture, Scroll, Text
MouseDown	Dir, File, Form, Label, List, Picture
MouseMove	Dir, File, Form, Label, List, Picture
MouseUp	Dir, File, Form, Label, List, Picture
Paint	Form, Picture
PathChange	File
PatternChange	File
Resize	Form
Timer	Timer
UnLoad	Form

that they are automatically invoked by the run-time system when the event eventName occurs at the object objectName. For example, consider the following sample code that handles clicking on a command button (with a Command1 control name and caption):

```
SUB Command1_Click()
    C$ = Command1.Caption ' save caption
    Command1.Caption = "You clicked me!"
    BEEP
    WaitForWhile 2000 ' wait for 2 seconds
```

```
    Command1.Caption = C$ ' restore caption
END SUB
```

When you click on the Command1 command button, the button recognizes the event and invokes the Command1_Click procedure to handle the click event. When the Command1_Click procedure ends, the event is declared handled. If there is no matching event-handling procedure, the event is abandoned. In the case of Command1_Click, the procedure displays a new caption for the button, beeps, waits for 2 seconds (the procedure WaitForWhile is assumed to be available in a module), and then restores the original button caption.

Event-handing procedures are similar to those for controls. A sample event-handling procedure for clicking on the form is shown below:

```
SUB Form_Click()
    BEEP
    WaitForWhile 100
    BEEP
END SUB
```

When you click on a clear area in the form, the form invokes the Form_Click procedure which results in two beeps.

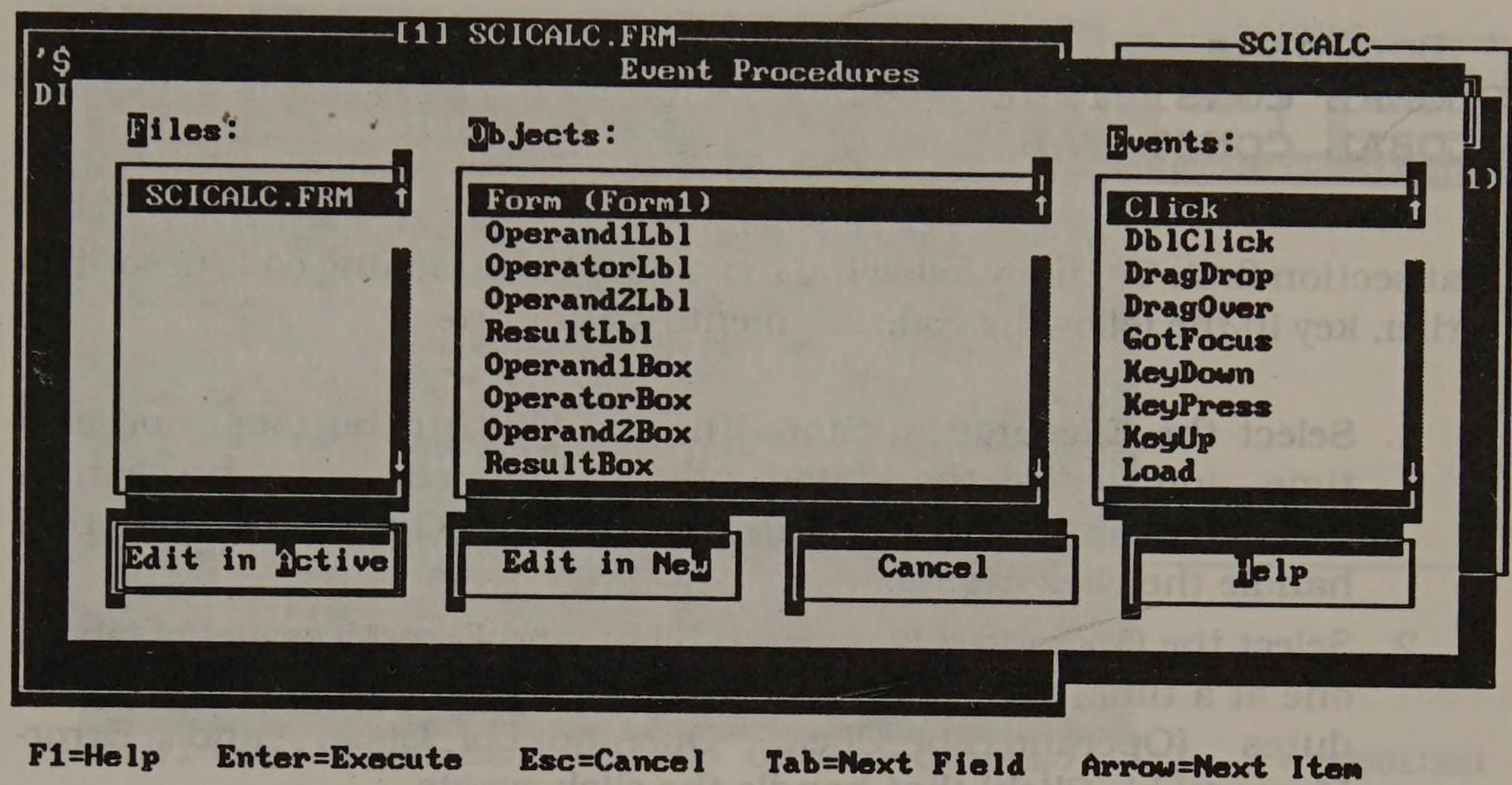
Working with the code window

Developing and attaching code for your Visual Basic application occurs in the Code window. This is a special window that makes it easier to write event-handling procedures. The features of the Code window include providing templates for the event-handling procedures and offering combo boxes that list the various objects and their events.

To access the Code window you have to be in the Programming environment. Select the Event Procedures option in the Edit menu. This option pops up a dialog box, shown in FIG. 6-1, that contains the Files list box, Objects list box, and Events list box. You first need to select your target object from the Objects list box.

Next, you need to select the event you want to handle. Double-click on the target event. Visual Basic responds by showing you a Code window that contains the event-handler SUB procedure. Initially, the SUB procedure you view has no statements. You can then type in the Basic statements that make up the body of the event-handling procedure. As with QuickBasic, the Visual Basic system checks, by default, the syntax of each line you type.

This syntax checking occurs when you either press the Enter key or move the editing cursor up or down. While you are on the same edited line, you can move the cursor back and forth without provoking the wrath of the syntax checker. You can turn off syntax checking by selecting the Syntax Checking option in the Options menu. The Edit menu offer the options for finding, replacing, copying, cutting, and pasting text. While editing a procedure, you can open a new Code window to view and/or edit another event-handler.



6-1 The Event Procedures dialog box.

Once you have finished coding the event-handling procedure(s) for the current object, you might want to move on to the next object. Select the Event Procedures option in the Edit menu and choose the next object.

Inserting code in CONSTANT.BI file

Before I discuss attaching the event-handling code to the various controls I want to focus on the contents of the CONSTANT.BI include file. Visual Basic uses .BI and .BAS extension names for include and module files, respectively. The CONSTANT.BI file contains frequently used constants, such as TRUE and FALSE. In addition, the file includes constants used by the Control Panel of Visual Basic. In many cases, especially when you are in the stage of learning to program with Visual Basic, you are only using a few constants in file CONSTANT.BI. You can create smaller files that contain only the constants you are actually using. This reduction makes your EXE programs smaller.

Rather than using the CONSTANT.BI file in its entirety, I created the SCICALC.BI file, shown in Listing 6-1. This included file declares the global constants TRUE and FALSE.

Attaching code to the controls

The next step in developing the COSC application is to code the various event-handling procedures. Listing 6-2 contains the code attached to the various controls, as well as the general declarations section. The default Code window contains the general declarations section. Type in the code for

Listing 6-1 The contents of the SCICALC.BI included file.

```
' Booleans
GLOBAL CONST TRUE = -1
GLOBAL CONST FALSE = 0
```

that section first. Then, by following the method of attaching code described earlier, key in the following code fragments (in any order):

1. Select the ExecuteBtn, StoreBtn, and QuitBtn buttons, one at a time, and key in the statements for their respective procedures (ExecuteBtn_Click, StoreBtn_Click, and QuitBtn_Click) that handle the click events.
2. Select the Operand1Lbl, Operand2Lbl, and ErrorMessageLbl labels, one at a time, and key in the statements for their respective procedures (Operand1Lbl_Click, Operand2Lbl_Click, and ErrorMessageLbl_Click) that handle the click events.

Figure 6-2 shows the code window displaying the code of an event-handler for the Execute command button.

Reading Listing 6-2 should be fairly easy. The majority of the code in that listing is easily recognized as event-handling procedures. The name of each procedure reflects the association between a control and an event it handles. Any code located at the beginning that is not part of an event-handling procedure comes from the general declarations section.

Now that I have presented the code attached to the SCICALC.FRM form, let's see how it works. I will discuss the procedures in the order in which they appear in the listing:

The DIM statement Declares a SHARED double-precision array, TheVars, with indices that range from 65 to 90. These numbers represent the ASCII code for the letters A and Z. Consequently, you can index the elements of array TheVars by simply providing the ASCII code of the single-letter variable name. The SHARED declaration is required to make the array accessible to all of the event-handling procedures in the form.

The ErrorMessageLbl_Click procedure Assigns a null string to the Text property of the Error Message text box.

The ExecuteBtn_Click procedure The heart of the command-oriented scientific calculator. In brief, the main purpose of this procedure is to convert the strings in the operand text box into numbers, read the string in the operator text box, perform the sought operation or function evaluation, and write the result in the Result box.

Obtaining numeric operands from the operand text boxes is complicated by the fact that they can contain the following kinds of data:

```
File  Edit  View  Search  Run  Debug  Options  Window                    Help
[1] SCICALC.FRM:ExecuteBtn_Click
SUB ExecuteBtn_Click ()
    DIM X AS SINGLE: DIM Y  AS SINGLE: DIM Z  AS DOUBL
    DIM Xchar AS STRING
    DIM YChar AS STRING
    ErrorMessageBox.Text = "" ' clear error message bo
    ResultBox.Text = ""       ' clear result box
      ' obtain the uppercase of the first character in
      ' Operand1Box text box
    Xchar = UCASE$(LEFT$(Operand1Box.Text, 1))
      ' obtain the uppercase of the first character in
      ' Operand2Box text box
    YChar = UCASE$(LEFT$(Operand2Box.Text, 1))
      ' convert string in Operand1Box text box to its
      ' numeric value.  Note that if the conversion fa
      ' X will be assigned a zero.
    X = VAL(Operand1Box.Text)
      ' is X = 0 when Xchar is not "0"?  This test det
      ' failed conversions
    IF (X = 0) AND (Xchar <> "0") THEN
<Shift+F1=Help> <F6=Window> <F2=Code> <F5=Run> <F8=Step>      N 00001:001
```

```
SCICALC
Form    Code
SCICALC.FRM  (Form1)
```

6-2 The Code window displaying procedure SUB ExecuteBtn_Click.

- The operand text box is empty.
- The string in the operand text box represents a valid number.
- The first character of the string in the operand text box is a letter.
- The first character of the string in the text box is neither a letter, nor a number, nor the + symbol, nor the − symbol.

To resolve these complications we start with the features of converting a string into a number as implemented by the VAL function. There are two of such conversions that occur at the following statements:

```
X = VAL(Operand1Box.Text)
Y = VAL(Operand2Box.Text)
```

The function returns a zero if it cannot convert at least the leading characters of a string into a number. Thus, when the Val function returns zero in the ExecuteBtn_Click procedure it could mean one of the following things:

- The operand text box has a string with zeros.
- The operand text box has a variable.
- The operand text box has an invalid number.
- The operand text box is empty.

Practically, the first case is the least likely one—you don't need a computer to carry out an operation that involves a zero! This means that the

code must check for a variable if the result of calling the Val function is 0 (and the first character in the text box is NOT "0"). The event-handling procedure uses a series of If statements to check if the uppercase of the first character of an operand text box is a letter. If this test is positive, the value of the variable is retrieved from the array TheVars by using the expressions TheVars(ASC(Xchar)) and TheVars(ASC(Ychar)).

The procedure uses a Select-Case statement to execute the sought math operation or function evaluation. The list of the math operators and functions include:

- Operators: +, −, *, /, ^, and = (assign operand 1 to the result)
- Functions: Ln, Log, Exp, Sqr, Sin, Cos, and Tan.

The Case clauses include argument/operand checking when appropriate. When a bad argument or operand is detected, the Error Message box displays an error message. The result of operation of function evaluation is stored in the local variable Z.

The last If statement in the procedure tests if the Error Message text box has remained empty (as initialized earlier in the procedure). An empty Error Message box indicates that there is no error. The statement in the accompanying Then clause converts the value of the variable Z into the new content of the Result text box.

The Operand1Lbl_Click and Operand2Lbl_Click procedures Perform very similar tasks—replacing the name of a variable with its contents. The code for these procedures examines the uppercase of the first character in an operand text box. If that character is a letter, the data for the corresponding variable is retrieved from the TheVars array. The string image of the number obtained is written to the operand text box.

The StoreBtn_Click Allows you to store the result of an operation in a variable. If the Variable box is empty, the procedure displays an error message in the Error Message box. Otherwise, the procedure performs the following steps:

- Obtains the uppercase of the first character in the Variable text box.
- Checks to determine if the extracted character is in the range A to Z.
- When the above test is positive, the ASCII code of the extracted character is assigned to the local variable Index. The contents of the Result text box are converted into a number and then stored in the array TheVars, using the variable Index as an index to the sought array element.
- The Error Message box is cleared to signal that the storage process is successful and also to remove any previous error message.

Listing 6-2 The code attached to the form and various controls.

```
'$INCLUDE: 'SCICALC.BI'

DIM SHARED TheVars(65 TO 90) AS DOUBLE

SUB ErrorMessageLbl_Click ()
      ErrorMessageBox.Text = ""
END SUB

SUB ExecuteBtn_Click ()
    DIM X AS SINGLE: DIM Y  AS SINGLE: DIM Z  AS DOUBLE
    DIM Xchar AS STRING
    DIM YChar AS STRING
    ErrorMessageBox.Text = "" ' clear error message box
    ResultBox.Text = ""       ' clear result box
    ' obtain the uppercase of the first character in the
    ' Operand1Box text box
    Xchar = UCASE$(LEFT$(Operand1Box.Text, 1))
    ' obtain the uppercase of the first character in the
    ' Operand2Box text box
    YChar = UCASE$(LEFT$(Operand2Box.Text, 1))
    ' convert string in Operand1Box text box to its
    ' numeric value.  Note that if the conversion fails,
    ' X will be assigned a zero.
    X = VAL(Operand1Box.Text)
    ' is X = 0 when Xchar is not "0"?  This test detects
    ' failed conversions
    IF (X = 0) AND (Xchar <> "0") THEN
        ' Yes!  Is Xchar in the range A to Z
        IF (Xchar >= "A") AND (Xchar <= "Z") THEN
            ' treat the content of Operand1Box as storing
            ' the name of a variable.  Assign the value of
            ' the variable to X
            X = TheVars#(ASC(Xchar))
        END IF
    END IF
    ' convert text in Operand2Box to numeric value
    Y = VAL(Operand2Box.Text)
    ' check if Operand2Box has a variable
    IF (Y = 0) AND (YChar <> "0") THEN
        IF (YChar >= "A") AND (YChar <= "Z") THEN
            ' Yes! there is a variable.  Assign its
            ' value to variable Y
            Y = TheVars#(ASC(YChar))
        END IF
    END IF
    ' examine the content of the OperatorBox text box
    SELECT CASE UCASE$(OperatorBox.Text)
    CASE "+"
```

Listing 6-2 Continued.

```
        Z = X + Y
    CASE "-"
        Z = X - Y
    CASE "*"
        Z = X * Y
    CASE "/"
        IF Y <> 0 THEN
            Z = X / Y
        ELSE
            ErrorMessageBox.Text = "Division by zero error"
            BEEP
        END IF
    CASE "^"
        Z = X ^ Y
    CASE "=" ' simply copy the value of the first
                 ' operand into the result box
        Z = X
    CASE "LN"
        IF X > 0 THEN
            Z = LOG(X)
        ELSE
            ErrorMessageBox.Text = "Bad function argument"
            BEEP
        END IF
    CASE "LOG"
        IF X > 0 THEN
            Z = LOG(X) / LOG(10)
        ELSE
            ErrorMessageBox.Text = "Bad function argument"
            BEEP
        END IF
    CASE "EXP"
        Z = EXP(X)
    CASE "SQR"
        IF X >= 0 THEN
            Z = SQR(X)
        ELSE
            ErrorMessageBox.Text = "Bad function argument"
            BEEP
        END IF
    CASE "SIN"
        Z = SIN(X * Pi / 180)
    CASE "COS"
        Z = SIN(X * Pi / 180)
    CASE "TAN"
        Z = TAN(X * Pi / 180)
    CASE ELSE
        ErrorMessageBox.Text = "Invalid operator"
        BEEP
END SELECT
```

```
        ' if the error message box is still empty, there is
        ' no error.  Therefore, convert the variable Z into
        ' a string and display it in ResultBox
        IF ErrorMessageBox.Text = "" THEN
            ResultBox.Text = STR$(Z)
        END IF
END SUB

SUB Label5_Click ()
    ' clear the error message text box
    ErrorMessageBox.Text = ""
END SUB

SUB Operand1Lbl_Click ()
    DIM Xchar AS STRING
    ' get the uppercase of the first character in the Operand1Box
    Xchar = UCASE$(LEFT$(Operand1Box.Text, 1))
    ' is the character in the range A to Z?
    IF (Xchar >= "A") AND (Xchar <= "Z") THEN
        ' Yes!  Obtain the variable from the array and write
        ' its numeric value to the text box
        Operand1Box.Text = STR$(TheVars#(ASC(Xchar)))
    END IF
END SUB

SUB Operand2Lbl_Click ()
    DIM Xchar AS STRING
    ' get the uppercase of the first character in the
    ' Operand2Box
    Xchar = UCASE$(LEFT$(Operand2Box.Text, 1))
    ' is the character in the range A to Z?
    IF (Xchar >= "A") AND (Xchar <= "Z") THEN
        ' Yes!  Obtain the variable from the array and write
        ' its numeric value to the text box
        Operand2Box.Text = STR$(TheVars#(ASC(Xchar)))
    END IF

END SUB

SUB QuitBtn_Click ()
    END
END SUB

SUB StoreBtn_Click ()
    DIM C AS STRING
    DIM Index AS INTEGER
    ' process the contents of the Variable text box
    IF VariableBox.Text <> "" THEN
        ' obtain the uppercase of the first character in the text box
        C = UCASE$(LEFT$(VariableBox.Text, 1))
        ' is the character in the range A to Z?
```

Listing 6-2 Continued.

```
IF (C >= "A") AND (C <= "Z") THEN
    ' Yes! Get the ASCII code for the character
    Index = ASC(C)
    ' store the value of the Result box in the array TheVars#
    TheVars#(Index) = VAL(ResultBox.Text)
    ' clear the error message box
    ErrorMessageBox.Text = ""
ELSE
    ' No! The text box has an invalid variable name
    ErrorMessageBox.Text = "Invalid variable name"
        BEEP
    END IF
ELSE
    ' the text box is empty!
    ErrorMessageBox.Text = "Missing variable name"
    BEEP
END IF
END SUB
```

The QuitBtn_Click procedure Uses an END statement to end the program execution.

Running the COSC application

To run the COSC application you can either press the F5 function key or select the Start option in the Run menu. If Visual Basic detects any errors it will display a window and highlight the offending code portion. If this occurs, check the code you typed and make the necessary corrections. Figure 6-3 shows the screen during a sample session with the COSC application.

Experiment with the various program features. If you feel comfortable with programming in Visual Basic, add a few more math functions or operators by expanding on the SELECT CASE statement in the ExecuteBtn_Click procedure. When you have finished, save the project files by using the "Save Project" option in the File menu.

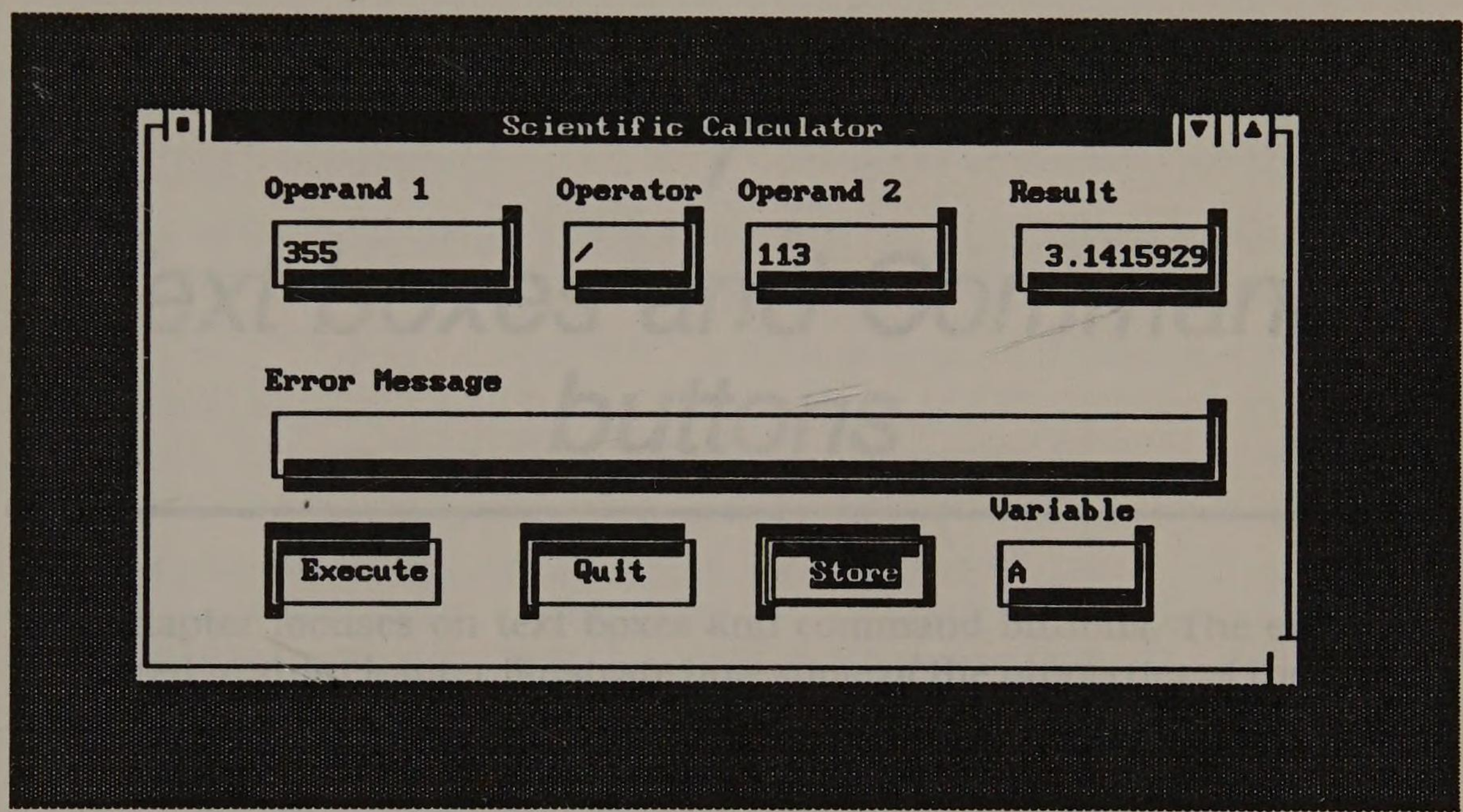
Creating an EXE file

Until now, you had to run Visual Basic in order to execute any Visual Basic program. Visual Basic enables you to create stand-alone .EXE files. The process of creating executable files is very easy and involves the following steps:

1. Open the project you need to compile into executable code.
2. Select the Make EXE File . . . option from the Run menu.
3. Enter the name of the executable file in the input box. Visual Basic supplies a default filename, based on the project filename, with an

.EXE extension. In the case of the COSC application, Visual Basic supplies you with the SCICALC.EXE filename.

4. Select the OK button to accept either the default filename or the different filename that you just typed. The executable file is written to the current directory, \VBDOS\EASYVB.



6-3 A sample session with the COSC application.

7

Text boxes and Command buttons

This chapter focuses on text boxes and command buttons. The examples presented in this chapter illustrate how some of the properties of these controls can be manipulated at run time.

Text boxes

In the previous chapters, I presented programs that use text boxes to type in a single-line input or display a single-line output. Among the Text box properties there are the following ones that prove to be interesting (at least for now):

The Multiline property This Boolean property enables a text box to display multiple lines of text. The Multiline property should be set to TRUE at design time. The sequence of CHR$(13) and CHR$(10) characters are used to visually separate the lines in a text box. For example, the following Visual Basic code stores an array of strings in a multiline text box, such that each string element appears on a separate line:

```
DIM F AS String
DIM NL AS String * 2
DIM I AS Integer
DIM A (1 TO 10) AS String
ReadTheStrings A() ' get the data from somewhere
F = ""
NL = CHR$(13) + CHR$(10)
FOR I = LBOUND(A, 1) TO UBOUND(A, 1)
    F = F + A(I) + NL
NEXT I
```

```
TextBox.Text = F
F = ""
```

The array A provides you with the strings. The variable NL is a two-character string that stores the CHR$(13) and CHR$(10) character sequence used to delimit text lines.

The variable F is used to sequentially build the large string that contains the members of the string array A, delimited by the NL strings. Once the variable F has been built, it is assigned to the Text property of TextBox. This technique produces a better visual effect—all of the lines in the text box appear at the same time.

The ScrollBars property This is an enumerated integer type that specifies whether or not a text box has scroll bars. The values for this property are:

0 - No scrollable bars appear with the text box.
1 - Horizontal scroll bar.
2 - Vertical scroll bar.
3 - Both scroll bars.

By setting the Multiline property to TRUE and setting the ScrollBars property to a non-zero value, Visual Basic practically turns a text box into a window that views and edits text. The keyboard cursor controls, such as HOME and END, are active, and the mouse can be used to select text for copying, deleting, and pasting. The first program in this chapter explores some of these powerful features.

The SelStart, SelLength, and SelText properties These properties, which are not available at design time, handle the selected text in text boxes.

The SelStart property sets or returns the starting point of the selected text. The valid range of values for SelStart is 0 to the length of the string in the text box. The value of SelStart indicates that the first character of the selected text is (SelStart + 1). If SelStart is assigned a value greater than the length of the text box string, Visual Basic automatically resets SelStart to the length of that string.

The SelLength property assigns or yields the size of the selected text; this is identical to the result of LEN(SelText). The valid range of values for SelLength is 0 to the length of the string in the text box. When SelStart changes values, SelLength is automatically assigned a 0. A run-time error occurs if a negative value is assigned to SelLength.

The SelText property assigns or returns the characters of the currently selected text. The SelText property contains an empty string when there is no selected text.

The general syntax for using the SelStart, SelLength, and SelText properties are shown below:

```
[form.][textBox.]SelStart [= index%]
[form.][textBox.]SelLength [= length%]
[form.][textBox.]SelText [= stringExpression$]
```

The Visible property This Boolean property empowers you to show or hide a text box. When the setting of the Visible property is TRUE (−1), the text box is visible. By contrast, if the setting is FALSE (0), the text box is invisible and inaccessible. The general syntax for the Visible property is:

[form.] [textBox.]Visible [= boolean%]

The first program, presented later in this chapter, uses all of the above text box properties.

Command buttons

All of the programs presented earlier in this book use command buttons to perform various tasks. This section focuses on the following command button properties:

The Visible property Command buttons have the Visible property which works just like with text boxes. The general syntax for using the Visible property with command buttons is given below:

[form.] [commandButton.]Visible [= boolean%]

The Enabled property This Boolean property empowers you to enable or disable a command button while still showing the button (unlike with using the Visible property). Setting the Enabled property to FALSE (0) disables a command button. This also affects the visual appearance of the button's caption—the caption of a disabled command button appears in a faint text font to indicate that the button is not in service.

Setting the Enabled property to TRUE (−1) activates a command button. The text for the button's caption returns to normal to indicate that the button is active. The general syntax for using the Enabled property is:

[form.] [commandButton.]Enabled [= boolean%]

The Default property This Boolean property specifies if a command button is the default button—the button that is initially selected when the host form is loaded. Pressing ENTER selects the default button. The general syntax for using the Default property is:

[form.] [commandButton.]Default [=boolean%]

The Cancel property This Boolean property that sets or determines if a command button is the Cancel button—the one automatically selected when you press the ESC key. Each form can have either one command button or none designated as the Cancel button. The Cancel button is selected at design time or run time. The general syntax for using the Cancel property is:

[form.] [commandButton.]Cancel [=boolean%]

The first program shown later in this chapter uses all of the above command button properties.

A text file viewer

The text box and command properties that appeared earlier in this chapter are the ingredients of a text file viewer application. The main purpose of the application, nicknamed FVIEW1, is to view ASCII text files. The secondary functionality of the program includes the following:

1. Perform a simple case-sensitive text search.
2. Invoke the DOS shell to execute a DOS command or run a DOS program.
3. Search for a file.

The specifications for the text file viewer application are listed in FIGS. 7-1, 7-2, and 7-3. Each figure represents a documentation form presented in chapters 4 and 5.

Figure 7-4 shows the form for the file viewer application with all of its controls, as specified in FIG. 7-3. Now that you have seen the program interface, let's discuss the functionality of the program in more detail.

The program's functionality

The main purpose of the program is to view ASCII text files. This is done by typing the name of a sought file (including the full directory path of that file) in the Filename Input text box. When the filename is entered, click on the Read button and the text lines of the specified file are displayed in the large text box. Use the scroll bars of the viewing text box to inspect the various parts of the file. The text box has its MultiLine and ScrollBars properties set to TRUE and 3, respectively, at design time. The result is a versatile file viewer.

Because the first task that you most likely want to perform is reading a file, the Default property of the Read button is set to TRUE. This means that when you start the program, you can type in the sought filename and press ENTER to view the text in that file. In fact, if you do not select any other command button, the above feature remains in effect. Once you select another command button, you then need to click on the Read button to read the specified file.

Text boxes are no match for the file list boxes when it comes to locating and selecting a file. Because file list boxes are introduced in a later chapter, we have to work with the text boxes for now. The Find File button performs a simple file search based on the DOS 5.0 dir command. To locate a file, type its name and the starting directory path (if this is omitted, the search begins from the current directory), then press the Find File button. If you are not running DOS 5.0, you need to use an alternate method. The technical aspects behind the Find File button are discussed in the next subsection.

The Toggle Read Button is a button that illustrates how to disable and enable a command button, specifically the Read button. Pressing the Toggle

Application Name: Text File Viewer	
Application Code Name: FVIEW1	
Version: 1.0	Date Created: June 27, 1992
Programmer(s): Namir Clement Shammas	
	List of Filenames
Storage Path:	C:\VB\EASYVB
Project	FVIEW1.MAK
Form 1	FVIEW1.FRM

7-1 General project specifications for the FVIEW1 project.

Form # 1	Form Filename: FVIEW1.FRM
Version: 1.0	Date: June 27, 1992

Control object type	Default CtlName	Purpose
Command Button	Command1	Reads the file specified in Text1.
	Command2	Finds the file specified in Text1.
	Command3	Toggles the Enabled setting of Command1.
	Command4	Finds the text in Text2.
	Command5	Executes a DOS command/program.
	Command6	Hides Command5, Text3, and Label3.
	Command7	Exits the application.
Text Box	Text1	Contains the filename to read.
	Text2	Contains the text search string.
	Text3	Contains the DOS command line.
	Text4	Displays the text lines of a file.
Label	Label1	Label of Text1.
	Label2	Label of Text2.
	Label3	Label of Text3.

7-2 List of control objects for the FVIEW1 project.

Read Button toggles the Enabled property of the Read button. When the Read button is disabled, clicking on it has no effect. You can tell whether or not the Read button is disabled by the style of font used in displaying the caption.

Once you have loaded a large file, you might want to search for text in

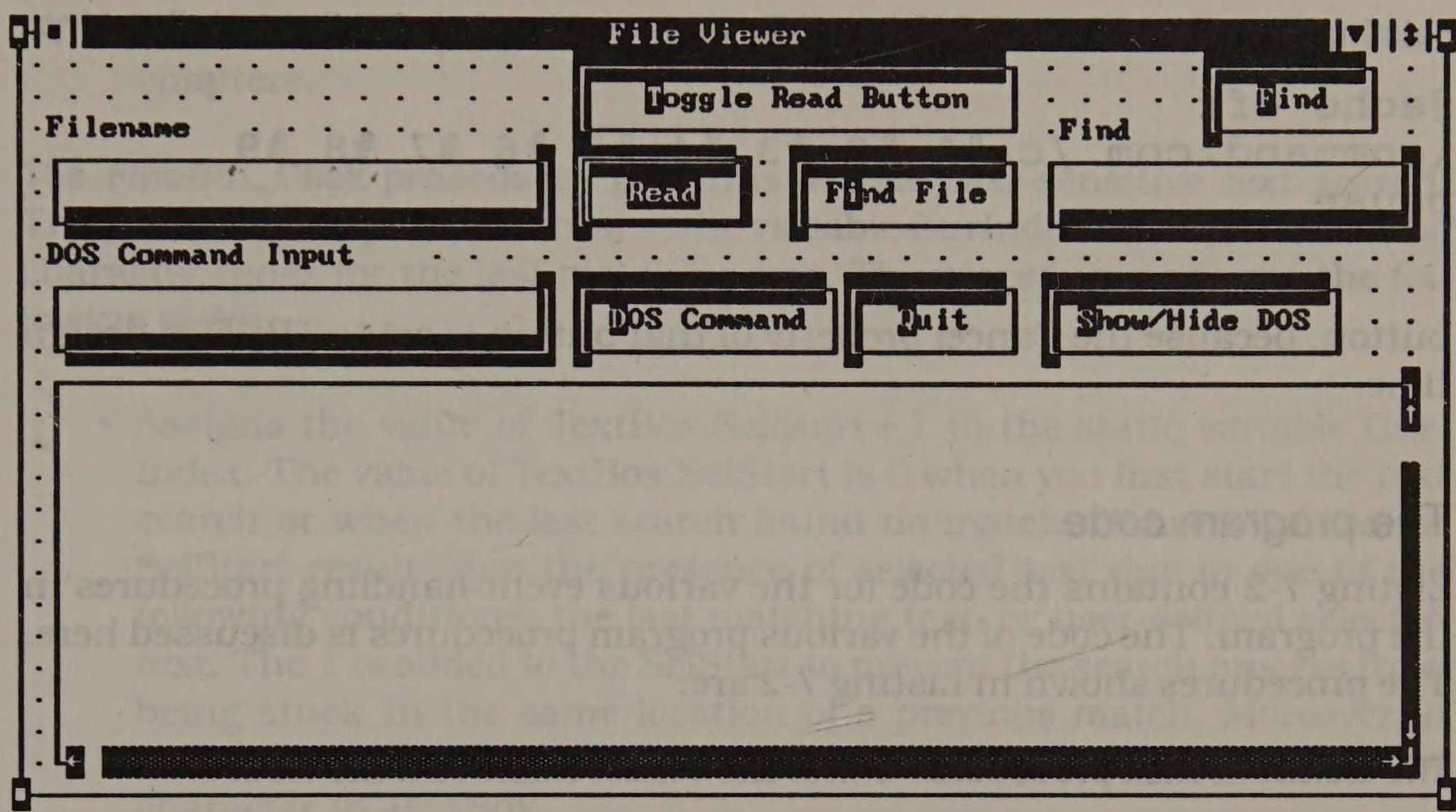
Application (Code) Name: FVIEW1
Form # 1
Version: 1.0 Date: June 27, 1992

Original control name	Property	New setting
Form	Caption	File Viewer
Command1	CtlName	ReadBtn
	Caption	Read
	Default	TRUE
Command2	CtlName	FindFileBtn
	Caption	Find File
Command3	CtlName	ToggleBtn
	Caption	Toggle Read Button
Command4	CtlName	FindBtn
	Caption	Find
Command5	CtlName	DosBtn
	Caption	DOS Command
Command6	CtlName	ShowOSBtn
	Caption	Show/Hide DOS
Command7	CtlName	QuitBtn
	Caption	Quit
	Cancel	TRUE
Text1	CtlName	FilenameBox
	Text	(empty string)
Text2	CtlName	FindBox
	Text	(empty string)
Text3	CtlName	DosBox
	Text	(empty string)
Text4	CtlName	TextBox
	Text	(empty string)
	MultiLine	TRUE
	ScrollBars	3 - both
Label1	CtlName	FilenameLbl
	Caption	Filename Input
Label2	CtlName	FindLbl
	Caption	Find Input
Label3	CtlName	DosLbl
	Caption	DOS Input

7-3 The modified control settings for the FVIEW1 project.

that file. The Find command button allows you to conduct a simple case-sensitive text search on the file currently in the text box. Type the search text in the Find Input text box and click the Find button to perform the text search. If the search is successful, the text box gets the focus and the matching text is displayed as selected text. When you click again on the Find button, the next matching text is located, and so on. If no match is found, the text box displays no selected text.

As coded, the FindBtn_Click event-handler is able to relocate matching



7-4 The form for the text file viewer program.

text in a cyclic fashion—after the last search found no text (assuming previous searches found a match), the next click on the Find button locates the first matching text in the file. By default, the search starts at the beginning of the file. You can direct the starting location of the search by selecting any arbitrary text (using the mouse) at that location.

The program also contains command buttons that empower you to run DOS commands and DOS applications. The DOS Command button handles the DOS commands and applications. Simply type the sought command or application in the DOS Input text box, and click the DOS Command button. The DOS Command button works with the DOSBAT.BAT batch file, shown in Listing 7-1. This file should be either in the current directory \VBDOS\EASYVB or any other directory that is part of the PATH environment variable. The DOSBAT.BAT batch file invokes the COMMAND.COM DOS command processor, assumed to be located in the root of the current drive.

If this is not the case in your system, you need to edit file DOSBAT.BAT to reflect the correct location of the COMMAND.COM file (or any compatible replacement). The PAUSE statement in the DOSBAT.BAT file allows for commands like the DOS DIR to pause after they finish working. Otherwise, the screen quickly switches back to the Visual Basic program, before you have a chance to view the last DOS screen generated by the DOS command you invoked.

The Show/Hide DOS command button toggles the visibility of the DOS Command button, the DOS Input text box, and the label of that box.

To exit the text file viewer application, you either press the ESC key or click on the Quit command button. The ESC key works to select the Quit

Listing 7-1 The DOSBAT.BAT batch file that must be placed at the root directory.

```
@echo off
\command.com /c %1 %2 %3 %4 %5 %6 %7 %8 %9
pause
```

button, because the Cancel property of that button is set to TRUE at design time.

The program code

Listing 7-2 contains the code for the various event-handling procedures in the program. The code of the various program procedures is discussed here. The procedures shown in Listing 7-2 are:

The QuitBtn_Click procedure Exits the Visual Basic text file view.

The ReadBtn_Click procedure Reads the text line of the file specified in the Filename Input text box. The procedure performs the following main tasks:

- Stores the Text setting of FilenameBox into the local variable F.
- Exits the procedure if the variable F contains an empty string.
- Assigns the sequence of CHR$(13) and CHR$(10) to the local variable NL.
- Sets the error-handler to trap the error that is caused by attempting to open a nonexistent file (as specified by variable F).
- Open the file F for input. The file buffer number (or handler, if you like) is 1.
- Assigns an empty string to the variable F. This variable is reused in storing the lines of the text files.
- Read the file text lines using the DO-WHILE loop. Each line read is stored in the local variable L. The variable F is appended to the L + NL string expression. This ensures that each line read is followed by a carriage return/line feed pair of characters.
- The variable F, which now stores the text lines, is assigned to the Text property of TextBox.
- The file buffer is closed and the procedure is exited.
- The remaining statements in the procedure form the error-handling statements. This includes the statement with the predefined MsgBox procedure that displays a "Cannot Open file filename" message. More information about the MsgBox procedure is detailed in chapter 9.

The ReadBtn_Click procedure performs file I/O and error-handling. These topics, which should be familiar to QuickBasic

and GW-Basic programmers, are discussed in more detail in later chapters.

The FindBtn_Click procedure Performs simple case-sensitive text search. The procedure employs the local static variable CurIndex to keep track of the character index for the last matching text. The procedure performs the following tasks:

- Assigns the value of TextBox.SelStart + 1 to the static variable CurIndex. The value of TextBox.SelStart is 0 when you first start the text search or when the last search found no match. Nonzero values for SelStart result from the presence of selected text due to one of the following conditions: the last matching text, or user-defined selected text. The 1 is added to the SelStart to prevent the search process from being stuck in the same location of a previous match. Moreover, if SelStart is 0, adding 1 makes the CurIndex variable point to the first character in TextBox.
- Assigns the string in the FindBox to the local string variable Find.
- Exits the procedure if the Find string is empty.
- Locates the occurrence of the next matching text using the predefined INSTR function. The call to INSTR includes the CurIndex argument to start the search at the CurIndex character of TextBox. Omitting the CurIndex argument results in the search always being conducted from the beginning of the file. The result of the INSTR function is reassigned to the CurIndex variable.
- Examines if the new value of CurIndex is positive. If so, there is a matching string found at character CurIndex. The first two statements in the THEN clause make the matching text the new selected text. The last statement in the THEN clause sets the focus to TextBox to make the selected text visible. If the CurIndex is zero, there is no matching text and the statements in the ELSE clause are executed to turn off any existing selected text. This action visually signals the absence of any matching text.

The DosBtn_Click procedure Allows you to execute a DOS command or DOS application. The procedure uses the Shell function to execute the DOSBAT.BAT file. The arguments for that batch file are provided by the DOS/Win Input text box. As shown in Listing 7-1, the DOSBAT.BAT file systematically invokes a copy of COMMAND.COM (the full filename is used). The /c option tells DOS that you want a copy of COMMAND.COM. The arguments you supply in the DOS Input box, such as "dir *.exe," specify exactly what you want DOS to do.

The FindFileBtn_Click A special procedure that searches for DOS files. This procedure is a special version of the DosBtn_Click procedure. The current

code for FindFileBtn_Click uses the new /s directive for the DOS 5.0 DIR command. The filename wildcard, obtained from the Filename Input text box, is inserted between the dir and /s/p arguments.

If you are not running DOS 5.0 or later, you need to change the first argument of the Shell function. If you have access to a file-finding utility, such as WhereIs or Peter Norton's FileFind, you can use either of the following statements:

```
Dummy = SHELL("dosbat.bat whereis " + FilenameBox.Text", 1)
Dummy = SHELL("dosbat.bat filefind " + FilenameBox.Text", 1)
```

The ToggleBtn_Click procedure Toggles the Enabled property of the Read command button. The procedure uses the Not operator in a simple assignment.

The ShowOSBtn_Click procedure Toggles the Visible property of the DOS Command button, the DOS Input text box, and the label of the latter text box. This procedure uses a series of simple assignments involving the NOT operator to reverse the Boolean value of the Visible property.

Listing 7-2 The listing for the FVIEW1 project.

```
'$INCLUDE: 'CONSTANT.BI'

SUB DosBtn_Click ()
    ' invoke the DOSBAT batch file
    SHELL ("dosbat.bat " + DosBox.Text)
END SUB

SUB FindBtn_Click ()
    STATIC CurIndex AS LONG
    DIM Find AS STRING
    ' update the index to the current character
    CurIndex = TextBox.SelStart + 1
    ' store the text of the Find box in the variable Find
    Find = FindBox.Text
    ' is variable Find empty?
    IF Find = "" THEN EXIT SUB ' nothing to find
    ' locate the index of the next substring Find in the
    ' text box
    CurIndex = INSTR(CurIndex + 1, TextBox.Text, Find)
    ' found a match?
    IF CurIndex > 0 THEN
        ' Yes! Display the matching text as selected text
        TextBox.SelStart = CurIndex - 1
        TextBox.SelLength = LEN(Find)
        TextBox.SETFOCUS
    ELSE
        ' No! Clear any selected text
```

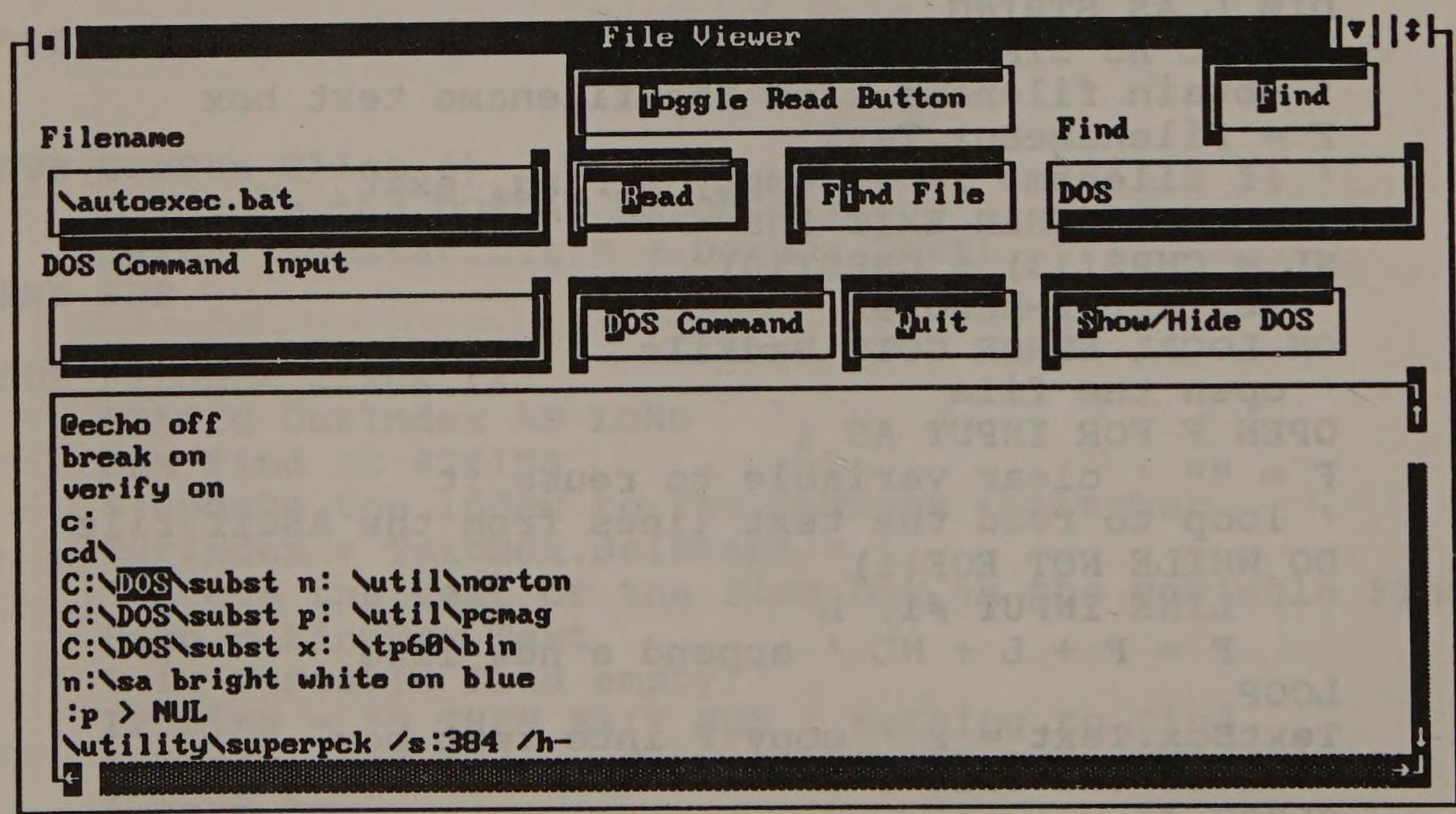
```
        TextBox.SelStart = 0
        TextBox.SelLength = 0
    END IF
END SUB

SUB FindFileBtn_Click ()
    ' invoke the DOS dir command with the /s
    ' subdirectory option and the /p paginated
    ' output option
    SHELL ("dosbat.bat dir " + FilenameBox.Text + " /s/p")
END SUB

SUB Form_Load ()
    WindowState = 2
END SUB

SUB QuitBtn_Click ()
    END
END SUB

SUB ReadBtn_Click ()
    DIM F AS STRING
    DIM L AS STRING
    DIM NL AS STRING * 2
    ' obtain filename from the filename text box
    F = FilenameBox.Text
    ' if filename is an empty string, exit
    IF F = "" THEN EXIT SUB
    NL = CHR$(13) + CHR$(10)
    ' set error-handler
    ON LOCAL ERROR GOTO BadFile
    ' open the file
    OPEN F FOR INPUT AS 1
    F = "" ' clear variable to reuse it
    ' loop to read the text lines from the ASCII file
    DO WHILE NOT EOF(1)
        LINE INPUT #1, L
        F = F + L + NL ' append a new line
    LOOP
    TextBox.Text = F ' copy F into text box
    ' close the file
    CLOSE #1
    ' exit procedure
    EXIT SUB
' ********** Error-handler *********
BadFile:
    BEEP
    MSGBOX "Cannot open file " + F, 0, "File I/O Error"
    RESUME EndOfSub
EndOfSub:
END SUB
```

Listing 7-2 Continued.

```
SUB ShowOSBtn_Click ()
    ' toggle the Visible property of the DosBox text box,
    ' the DosBtn button, the WinBtn button, and the DosLbl
    ' label
    DosBox.Visible = NOT DosBox.Visible
    DosBtn.Visible = NOT DosBtn.Visible
    WinBtn.Visible = NOT WinBtn.Visible
    DosLbl.Visible = NOT DosLbl.Visible
END SUB

SUB ToggleBtn_Click ()
    ReadBtn.Enabled = NOT ReadBtn.Enabled
END SUB
```

Running the file viewer

Once you have typed the code for the file viewer, or loaded from the disk, press the F5 function key to run the program. A sample session of using the text file viewer is shown in FIG. 7-5.



7-5 A sample session with the text file viewer program.

Setting access keys to command buttons

Access keys empower you to select a command button by pressing the ALT key with the designated character for that button. The designated character is the underlined character that appears as part of the button's caption. If a

Object	Property	New setting
Form	Caption	File Viewer (1B)
Command1	Caption	&Read
Command2	Caption	F&ind File
Command3	Caption	&Toggle Read Button
Command4	Caption	&Find
Command5	Caption	&DOS Command
Command6	Caption	&Show/Hide OS
Command7	Caption	&Quit

7-6 The updated settings for the text file viewer that introduces access keys to the command buttons.

command button has no character that is underlined, it has no access key associated with it.

To assign an access key to a command button, place the ampersand character, &, before the designated character in the caption. The ampersand character itself does not become part of the visible caption—it is merely an access-key selector. For example, to make the ALT-R the access key to the Read button in the text file viewer, make the caption of that button &Read. Access keys are not case-sensitive and need not be the first character, but they must be unique.

Figure 7-6 shows the updated settings for the text file viewer that assign access keys to the control buttons. The accompanying disk stores the updated form and project files under FVIEW1B.FRM and FVIEW1B.MAK, respectively.

Manipulating the tab order of controls

Visual Basic keeps track of the order in which you create the various controls on your form. By default, Visual Basic uses that same order when you press the TAB key to select the next control. This is known as the tab order. Thus, a well-planned and well-created form enables you to smoothly navigate between neighboring (either visually or logically) controls. By contrast, a hastily assembled form causes the TAB key to jump erratically between controls. Fortunately, Visual Basic allows you to override that default tab order.

Many controls possess the TabStop property, which is usually set to TRUE. This value indicates that the control can be selected when you cycle through the controls using the TAB key. Setting TabStop to FALSE makes Visual Basic bypass that control when you TAB through the form controls.

The controls that have the TabStop property also possess the TabIndex property. This is the property that Visual Basic uses in determining the tab order. You can view the TabIndex setting for each control by selecting the TabIndex property in the properties combo box. The lowest value for the TabIndex is 0. Negative TabIndex values generate an error.

Changing the tab order can be made either during design time or during run time. You can type in a new tab order value for the currently selected control. Visual Basic automatically adjusts the TabIndex setting for the controls that have the same setting or greater—the TabIndex of these controls is incremented by one. If you type in a tab order that is higher than the upper limit, Visual Basic adjusts that value to the upper limit. This upper limit is equal to the total number of controls in the tab order minus one.

Altering the tab order during run time is carried out by assigning a new value to the TabIndex property. The best place to insert the new tab order setting is the Form_Load procedure.

☞ Emulating scrollable DOS screens

This section explores one basic programming method that allows you to emulate an extended version of the PRINT statement in a text box and treat the text box as a pseudoscreen emulating a scrollable DOS screen. Concerning the task of emulating a scrollable DOS screen, there is some good news and some bad news. First the bad news—the method presented here causes the emulated screens to show text slower than ordinary screens. The good news is that you can have screens of various sizes (rows and columns) and you can show multiple emulated screens in the same application. An added bonus is that the emulated screens can be scrollable.

The next subsections discuss the basic programming approach in designing an emulated scrollable DOS screen, show two versions of the code, and present a Visual Basic program that tests the simpler code version.

The basic approach

The concept of emulating a scrollable DOS screen (which will be referred to as a pseudoscreen from now on) builds on the ability of text boxes to display multiple lines of text. The design of pseudoscreens involves the following components and features:

- The pseudoscreen has a user-defined size—maximum number of rows and columns. This helps to keep text in check.
- The pseudoscreen has a hidden cursor that keeps track of the location of the text displayed the next time you want to write to the screen.
- You can move the cursor and query its location. The cursor is bound by the pseudoscreen dimensions.
- You can write a string at the current screen location using a pseudo-PRINT procedure. The text can wrap around to, at least, the next line. The pseudo-PRINT emulates a QuickBasic "PRINT stringExpression$;" statement and thus excludes moving the cursor

to the next line (unless the text wraps). A separate procedure is used to move the cursor to the next line.

The above specifications describe the general behavior of the pseudoscreens, but sheds no light on the basic approach for the implementation. The implementation involves a sequence of design choices. The first choice is whether or not to directly update the text lines in the text box. This update requires you to read and write the substrings that form individual lines in the multiline text box.

Because Visual Basic does not implement methods to perform the above type of substring accesses, the task can be slow and involved. The second design choice is to maintain a copy of the text lines in an array of dynamic strings. The changes to the screen are made in two quick steps: first update the array elements, and then copy the array elements to the screen to refresh the display. Whereas this approach uses twice as much memory to display the text, it is easier to code. The ease of coding is due to the fact that using arrays permits you to apply the divide-and-conquer programming principle, and deal with each line separately. In addition, you can write to several lines on the pseudoscreen before updating the display.

When using arrays of strings to maintain a copy of the pseudoscreen, you have two choices of how to manipulate the text in the array elements. The first choice initializes the array elements with spaces—each element has a string of spaces equal to the maximum number of screen columns. Consequently, writing to the pseudoscreen involves overwriting the characters (initially the spaces) in the array elements. In addition, operations like clearing the screens involve overwriting the array elements with spaces.

The second choice in using array elements is more memory conservative than the fist one; after all, these arrays already duplicate the viewed text lines. The array elements are expanded only as needed. This expansion is due to the use of a pseudo-PRINT command and moving the cursor. In the latter case, the array element of the sought line is padded with spaces up to the sought column position of the cursor.

Another design choice for the pseudoscreen involves how general the code should be. At one extreme, the code handles a single pseudoscreen and its associated array of strings. At the other extreme, the code handles any text box associated with at least one array of strings. Both extremes are presented in this section.

The code for the first pseudoscreen

Let's start with the simple case where a single pseudoscreen is implemented using a text box and an array of strings. Listing 7-3 contains the code for the procedures that manipulate the pseudoscreen. The code is located in the declarations section and has access to the ScreenBox text box, which is used to emulate a scrollable DOS screen. The general section declares the

MAX_ROWS and MAX_COLS constants to specify the size of the pseudoscreen. In addition, the same code section declares the string array, ScrLine, the CursX variable, and the CursY variable. The last two variables store the current cursor location. There are other variables that are also declared in that section. I will explain these variables a little bit later in this subsection.

The procedures that manipulate the pseudoscreen supported by object ScreenBox, array ScrLine, and variables CursX and CursY are:

The ClearScreen procedure Clears the pseudoscreen. This involves assigning empty strings to the ScreenBox and the elements of array ScrLine. In addition, the cursor location is set to the upper-right corner of the pseudoscreen, (1, 1), by assigning 1 to the cursor variables.

The GotoXY procedure Moves the cursor to the screen location (X, Y). The procedure checks if the provided coordinates are valid. The array element of the Y row is padded with spaces if the specified X coordinate is greater than the length of that array element. The valid arguments for X and Y are assigned to the cursor variables, CursX and CursY.

The WhereX function Returns the value of the variable CursX.

The WhereY function Returns the value of the variable CursY.

The ScrollUp procedure Emulates scrolling the pseudoscreen a specified number of lines. Two FOR-NEXT loops are used to shift/copy text among the elements of array ScrLine before the pseudoscreen is updated.

The NewLine procedure Places the cursor on the next line. If the value of CursY is equal to MAX_ROWS, the pseudoscreen is scrolled up by one line.

The PPrint procedure Implements the extended version of PRINT mentioned earlier. The current implementation allows PPrint to display text that can wrap only to the next line.

The SaveScreen procedure Saves the present pseudoscreen and the current cursor position to the parameters Buffer(), X%, and Y%, respectively. Later, you can call the LoadScreen to restore the screen. The SaveScreen and LoadScreen procedures allow you to take snapshot images of the pseudoscreen and to view these images later. The SaveScreen procedure simply copies the ScrLine array to the Buffer array, variable CursX to X%, and variable CursY to Y%.

The LoadScreen Loads the pseudoscreen from the data in the arguments for the Buffer, X%, and Y%. The LoadScreen first copies the Buffer array to the ScrLine array, X% to variable CursX, and Y% to variable CursY. Once the data is copied, the pseudoscreen is updated.

The UpdateScreenText procedure Updates the ScreenBox to match the strings of array ScrLine.

Listing 7-3 The code for the PSCREEN project.

```
DECLARE SUB ClearScreen ()
DECLARE SUB GotoXY (X AS INTEGER, Y AS INTEGER)
DECLARE FUNCTION WhereX () AS INTEGER
DECLARE FUNCTION WhereY () AS INTEGER
DECLARE SUB ScrollUp (NumLines AS INTEGER)
DECLARE SUB NewLine ()
DECLARE SUB PPrint (S AS STRING, UpdateScreenNow AS INTEGER)
DECLARE SUB SaveScreen (Buffer() AS STRING, X AS INTEGER, Y AS INTEGER)
DECLARE SUB LoadScreen (Buffer() AS STRING, X AS INTEGER, Y AS INTEGER)
DECLARE SUB UpdateScreenText ()
DECLARE SUB LocateBtn_Click ()
' constants the define the size of the emulated screen
CONST MAX_ROWS = 25, MAX_COLS = 80
' the array and variables that maintain the data for
' the emulated screen
DIM SHARED ScrLine(1 TO MAX_ROWS)  AS STRING
DIM SHARED CursX AS INTEGER
DIM SHARED CursY AS INTEGER
' declarations for the screen buffer
DIM SHARED ScreenBuffer(1 TO MAX_ROWS) AS STRING
DIM SHARED BufX AS INTEGER
DIM SHARED BufY AS INTEGER

SUB ClearBtn_Click ()
    ClearScreen
    LocateBtn_Click
END SUB

SUB ClearScreen ()
    ' clear the screen by assigning empty strings to the
    ' ScreenBox and ScrLine array and setting the cursor
    ' variables to 1.
    DIM I AS INTEGER
    ScreenBox.Text = ""
    FOR I = 1 TO MAX_ROWS
        ScrLine(I) = ""
    NEXT I
    CursX = 1
    CursY = 1
END SUB

SUB Form_Load ()
    WindowState = 2
    ClearScreen
END SUB

SUB GotoXY (X AS INTEGER, Y AS INTEGER)
    ' move the hidden cursor to (X,Y)
    DIM L AS INTEGER
    IF (X < 1) OR (Y < 1) THEN EXIT SUB
    IF (Y > MAX_ROWS) OR (X > MAX_COLS) THEN EXIT SUB
    L = LEN(ScrLine(Y))
    IF X > L THEN ' Pad string with spaces
        ScrLine(Y) = ScrLine(Y) + SPACE$(X - L)
    END IF
    CursX = X
    CursY = Y
END SUB

SUB LoadBtn_Click ()
    LoadScreen ScreenBuffer(), BufX, BufY
```

Listing 7-3 Continued.

```
    LocateBtn_Click
END SUB

SUB LoadScreen (Buffer() AS STRING, X AS INTEGER, Y AS INTEGER)
    ' load screen from the Buffer() array
    ' the X% and Y% parameters specify new cursor location
    DIM I AS INTEGER
    FOR I = 1 TO MAX_ROWS
        ScrLine(I) = Buffer(I)
    NEXT I
    CursX = X%
    CursY = Y%
    UpdateScreenText
END SUB

SUB LocateBtn_Click ()
    CursorXBox.Text = STR$(WhereX())
    CursorYBox.Text = STR$(WhereY())
END SUB

SUB MoveBtn_Click ()
    DIM X AS INTEGER
    DIM Y AS INTEGER
    X = VAL(CursorXBox.Text)
    Y = VAL(CursorYBox.Text)
    GotoXY X, Y
    LocateBtn_Click
END SUB

SUB NewLine ()
    ' move the hidden cursor to the first column of
    ' the next line.  Scroll screen up if the cursor
    ' is already at the last allowed screen row
    IF CursY < MAX_ROWS THEN
        CursY = CursY + 1
        CursX = 1
    ELSE
        ScrollUp 1
        CursX = 1
    END IF
END SUB

SUB NewLineBtn_Click ()
    NewLine
    LocateBtn_Click
END SUB

SUB PPrint (S AS STRING, UpdateScreenNow AS INTEGER)
    ' Emulate a simple form of the QuickBasic print:
    '
    '             PRINT Astring$;
    '
    ' The second parameter enable you to update the text
    ' on the screen, or keep the changes hidden (for now).
    DIM LenStr AS INTEGER
    DIM LenLine AS INTEGER
    DIM LenDiff AS INTEGER
    DIM S2 AS STRING
    IF S = "" THEN EXIT SUB
    LenStr = LEN(S)
    IF CursY = MAX_ROWS THEN ScrollUp 1
```

```
    LenLine = LEN(ScrLine(CursY))
    S2 = ""
    ' string cannot fit on the current line?
    IF (CursX + LenStr) > MAX_COLS THEN
        LenDiff = CursX + LenStr - MAX_COLS - 1
        ' split original string into two strings
        S2 = RIGHT$(S, LenDiff) ' next-line text
        S = LEFT$(S, LenStr - LenDiff)
    END IF
    ' Pad current line
    IF (CursX + LenStr) > LenLine THEN
        LenDiff = CursX + LenStr - LenLine
        ScrLine(CursY) = ScrLine(CursY) + SPACE$(LenDiff)
    END IF
    ' write S to current line
    MID$(ScrLine(CursY), CursX, LenStr) = S
    CursX = CursX + LenStr ' update CursX
    IF CursX > MAX_COLS THEN NewLine
    ' the next-line string is not empty?
    IF S2 <> "" THEN ' print to the next line
        IF CursY < MAX_COLS THEN NewLine
        LenDiff = LEN(S2) - LEN(ScrLine(CursY))
        IF LenDiff > 0 THEN ' pad the string for the next line
            ScrLine(CursY) = ScrLine(CursY) + SPACE$(LenDiff)
        END IF
        ' write the next-line string
        MID$(ScrLine(CursY), 1, LEN(S2)) = S2
        CursX = LEN(S2) + 1
        IF CursX > MAX_COLS THEN NewLine
    END IF
    ' update the screen now?
    IF UpdateScreenNow THEN UpdateScreenText
    END SUB

    SUB PrintBtn_Click ()
        DIM S AS STRING
        S = TextBox.Text
        PPrint S, -1
        LocateBtn_Click
    END SUB

    SUB QuitBtn_Click ()
            END
    END SUB

    SUB SaveBtn_Click ()
        SaveScreen ScreenBuffer(), BufX, BufY
    END SUB

    SUB SaveScreen (Buffer() AS STRING, X AS INTEGER, Y AS INTEGER)
        ' save screen to Buffer() array.
        ' the current position of the hidden cursor is
        ' stored in parameters X% and Y%
        DIM I AS INTEGER
        FOR I = 1 TO MAX_ROWS
            Buffer(I) = ScrLine(I)
        NEXT I
        X% = CursX
        Y% = CursY
    END SUB

    SUB ScrollUp (NumLines AS INTEGER)
        ' scroll up a specified number of lines
        DIM I AS INTEGER
```

Listing 7-3 Continued.

```
IF NumLines < 1 THEN EXIT SUB
' scroll at most MAX_ROWS rows
IF NumLines > MAX_ROWS THEN
    NumLines = MAX_ROWS
END IF
' copy leading string to emulate scroll
FOR I = 1 TO MAX_ROWS - NumLines
    ScrLine(I) = ScrLine(I + NumLines)
NEXT I
' assign empty string to trailing strings
FOR I = MAX_ROWS - NumLines + 1 TO MAX_ROWS
    ScrLine(I) = ""
    NEXT I
    UpdateScreenText
END SUB

SUB ScrollUpBtn_Click ()
    ScrollUp 1
    LocateBtn_Click
END SUB

SUB UpdateScreenText ()
    ' update the text in the ScreenBox
    DIM I AS INTEGER
    DIM S AS STRING
    DIM NL AS STRING * 2
    NL = CHR$(13) + CHR$(10)
    S = ""
    FOR I = 1 TO MAX_ROWS - 1
        S = S + ScrLine(I) + NL
    NEXT I
    S = S + ScrLine(MAX_ROWS)
    ScreenBox.Text = S
END SUB

FUNCTION WhereX () AS INTEGER
    ' return the value of CursX
    WhereX = CursX
END FUNCTION

FUNCTION WhereY () AS INTEGER
    ' return the value of CursY
    WhereY = CursY
END FUNCTION
```

The PSCREEN project

Listing 7-3 also contains event-handling procedures for a Visual Basic application that allows you to test the above screen-manipulation procedures. The test application uses the PSCREEN.MAK project file and the PSCREEN.FRM form file.

Figures 7-7, 7-8, and 7-9 contain the documentation forms that specify the various objects and their customized settings. Figure 7-10 shows the controls of the test program during design time. The various command buttons enable you to test various aspects of the pseudoscreen, such as, clear-

Application Name: Pseudoscreen Test Program	
Application Code Name: PSCREEN	
Version: 1.0	Date Created: June 28, 1992
Programmer(s): Namir Clement Shammas	
	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	PSCREEN.MAK
Form 1	PSCREEN.FRM

7-7 General project specifications for the PSCREEN project.

ing the screen, moving the cursor, printing text, scrolling up the screen, saving the screen, and restoring the screen.

Listing 7-3 contains the code for the various event-handling procedures. The Form_Load procedure clears the pseudoscreen when the PSCREEN application starts running.

The Locate Cursor button is actually invoked by the other procedure-handling events to automatically display the updated location of the cursor in the text boxes marked Column and Row. This should compensate for the fact that the cursor is invisible. It is important to point out that the

Form # 1	Form Filename: PSCREEN.FRM
Version: 1.0	Date: June 28, 1992

Control object type	Default CtlName	Purpose
Command Button	Command1	Exits the test program
	Command2	Clears the pseudoscreen.
	Command3	Moves the cursor.
	Command4	Locates the cursor.
	Command5	Writes the text in Text4 to the pseudoscreen (Text1).
	Command6	Scrolls up the pseudoscreen.
	Command7	Moves the cursor to the next line.
	Command8	Saves the pseudoscreen.
	Command9	Restore the pseudoscreen.
Text Box	Text1	The pseudoscreen.
	Text2	The cursor column number.
	Text3	The cursor row number.
	Text4	Text to write on the pseudoscreen.
Label	Label1	Label for Text1.
	Label2	Label for Text2.
	Label3	Label for Text3.
	Label4	Label for Text4.

7-8 List of control objects for the PSCREEN project.

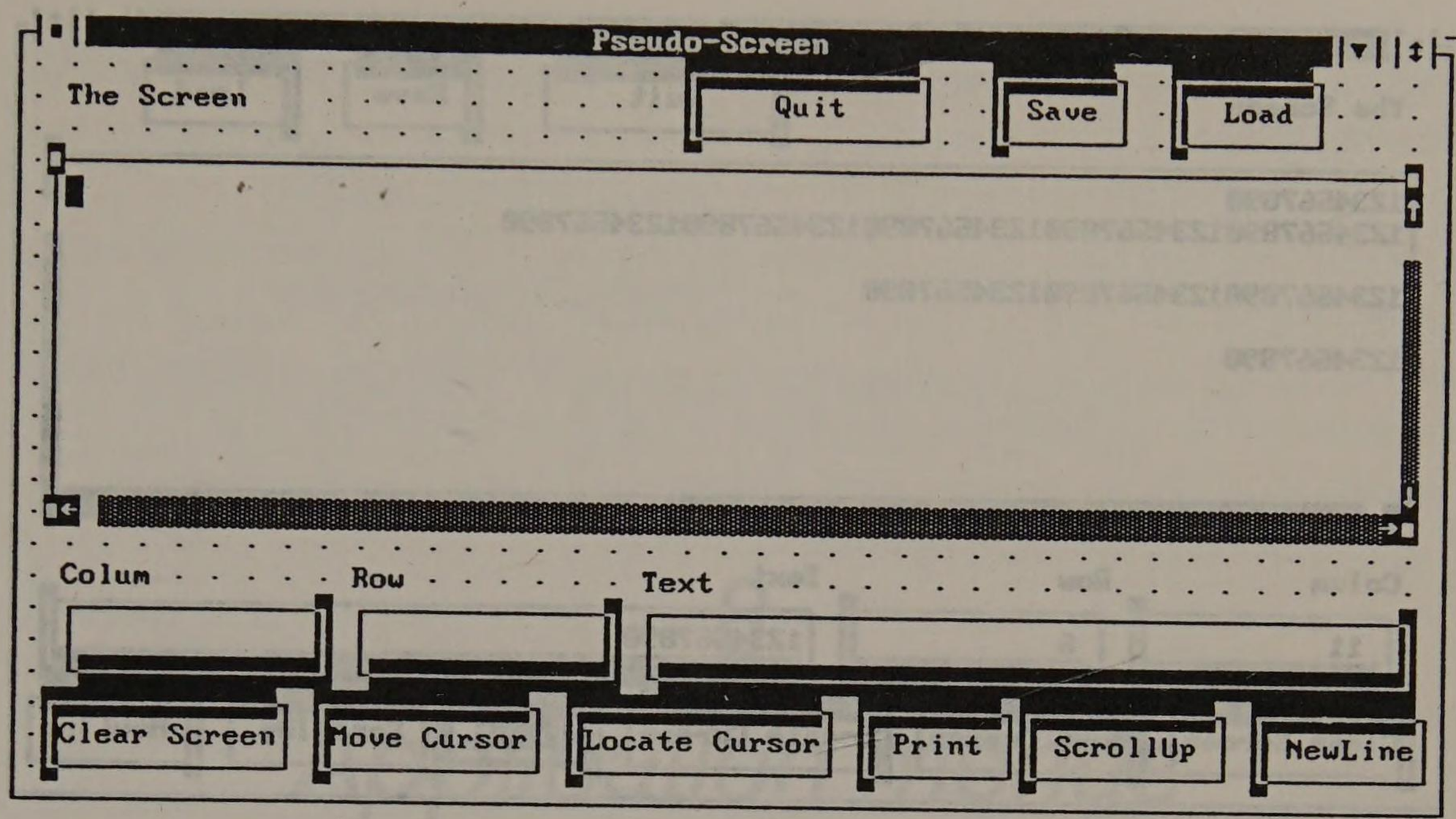
Application (Code) Name: PSCREEN
Form # 1
Version: 1.0 Date: June 28, 1992

Original control name	Property	New setting
Form	Caption	Pseudoscreen
Command1	CtlName	QuitBtn
	Caption	Quit
Command2	CtlName	ClearBtn
	Caption	Clear screen
Command3	CtlName	MoveBtn
	Caption	Move cursor
Command4	CtlName	LocateBtn
	Caption	Locate cursor
Command5	CtlName	PrintBtn
	Caption	Print
Command6	CtlName	ScrollBtn
	Caption	Scroll Up
Command7	CtlName	NewLineBtn
	Caption	NewLine
Command8	CtlName	SaveBtn
	Caption	Save
Command9	CtlName	LoadBtn
	Caption	Load
Text1	CtlName	ScreenBox
	Text	(empty string)
	MultiLine	TRUE
	ScrollBars	3 - both
Text2	CtlName	CursorXBox
	Text	(empty string)
Text3	CtlName	CursorYBox
	Text	(empty string)
Text4	CtlName	TextBox
	Text	(empty string)
Label1	CtlName	ScreenLbl
	Caption	The screen
Label2	CtlName	ColumnLbl
	Caption	Column
Label3	CtlName	RowLbl
	Caption	Row
Label4	CtlName	TextLbl
	Caption	Text

7-9 The modified control settings for the PSCREEN project.

pseudoscreen cursor has nothing to do with the edit cursor that appears when you click in the ScreenBox.

The general declarations section declares the following array and vari-



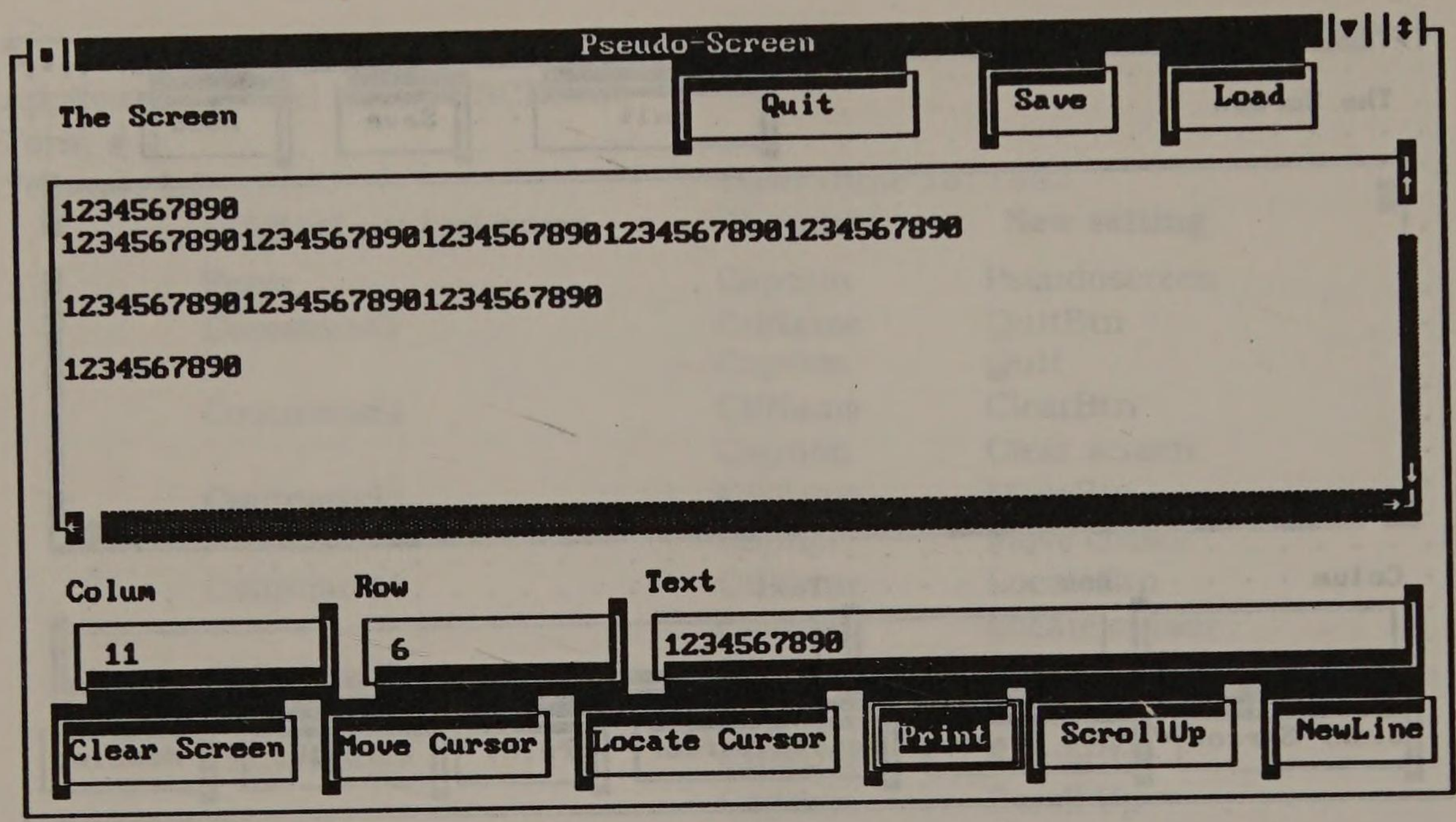
7-10 The control objects of the PSCREEN test application.

ables that are used in testing the SaveScreen and LoadScreen procedures:

```
' declarations for the screen buffer
DIM ScreenBuffer(1 TO MAX_ROWS) AS STRING
DIM BufX AS INTEGER
DIM BufY AS INTEGER
```

The PSCREEN test program uses the above array and variables as its sole buffer that stores the current pseudoscreen. You can extend the number of buffers used by the program. This requires additional arrays and variables, similar to ScreenBuffer, BufX, and BufY, as well as additional controls to select the target buffer.

After entering the code in Listing 7-3, or loading it from disk, run the PSCREEN test program by pressing the F5 function key. Type in some text in the Text text box and then press the Print command button. Try pressing the NewLine button and then press the Print button again. Notice the text on the pseudoscreen. The Row and Column text boxes update the current location of the hidden cursor. Experiment with the various controls to get a feel of pseudoscreen implementation. Figure 7-11 shows the screen image of a sample session with the PSCREEN test program.



7-11 A sample session with the PSCREEN test program.

8
Application menus

Menus are an integral part of the many DOS applications and utilities. Until now, the Visual Basic programs that I presented did not build any custom menus. Instead, they only had the "application" menu that is automatically attached to every Visual Basic program. Incorporating custom menus in your Visual Basic applications is an easy process. The designers of Visual Basic offer you a powerful menu builder that makes it easy to build and to modify custom menus.

☞ Definitions

This book uses the following terms to identify menu components:

Menu selection A member of the main menu.

Menu option A member of the choices offered by a menu selection, or a parent menu option (in the case of nested menus).

Menu item A collective name for both a menu selection and a menu option.

The Menu Design Window

Creating a custom application menu is easy with the Menu Design Window. Visual Basic treats the menus selections and options as controls that recognize the Click event. Due to the nature of menus, the designers of Visual Basic have elected to use a specialized window in building the menus instead of using the Toolbox. Figure 8-1 shows the Menu Design Window while building a sample application menu. The Menu Design Window contains the following controls:

The Caption text box This stores the caption for a menu option or selection. The hyphen character, -, is a special caption. When a menu item is assigned the hyphen caption it appears as a separator bar that divides the various menu options into logical groups.

The CtlName text box This contains the control name of a menu option or selection.

The Index text box This stores the index number for a menu option that is a control array. (More about control arrays later in this chapter.)

The Shortcut Key combo box This combo box enables you to assign shortcut keys for the various menu options.

The Checked, Enabled, Visible, Separator check boxes These boxes represent Boolean properties for menu selections and options, and allow you to establish the initial Boolean values for these properties. The Checked property determines whether or not a check mark appears to the left of a menu selection or option, to inform the user of the Boolean state associated with that menu item. The Enabled property allows you to enable or disable a menu item. Disabling a menu selection, for example, disables all of its options. Disabled menu items are displayed in lighter and shaded fonts. The Visible property permits you to show or hide a menu item. Setting a Visible property of a menu item to TRUE both disables that item and hides it. The Separator check box offers a second method for creating separators.

The Done command button Click on this button when you are finished with designing the menu. You can always return to the Menu Design Window and update the menu design.

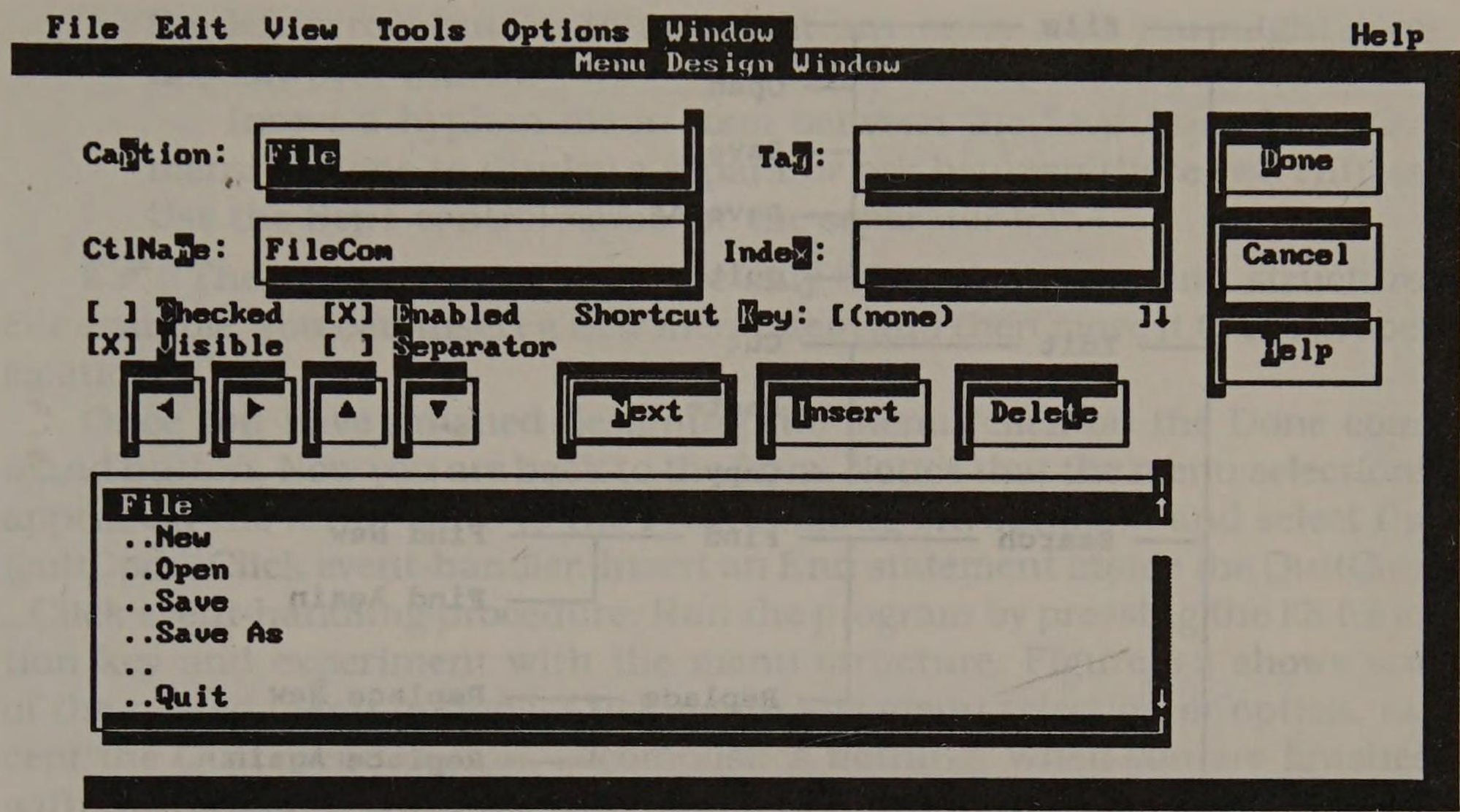
The Cancel command button This button cancels the updates you have made in the current session with the Menu Design Window.

The Help command button This button provides you with online help on topics related to the menu design.

The menu structure text box This is a scrollable text box that displays the various menu items. The vertical scroll bar appears when the text box cannot display all of the menu items. The current menu item is displayed in reverse video.

The menu-building command buttons These are the command buttons located right above the menu structure text box. The Delete button removes the current menu item. The Insert button inserts a new menu item just before the current menu item. The Next button enables you to select the next menu item. The up and down arrow buttons allow you to move the current selection up or down the list of menu items, respectively. The left and right arrow buttons play an important role in creating the menu structure. (More about these important buttons later in this section.)

The process of creating a menu structure is essentially one of creating



8-1 The Menu Design Window.

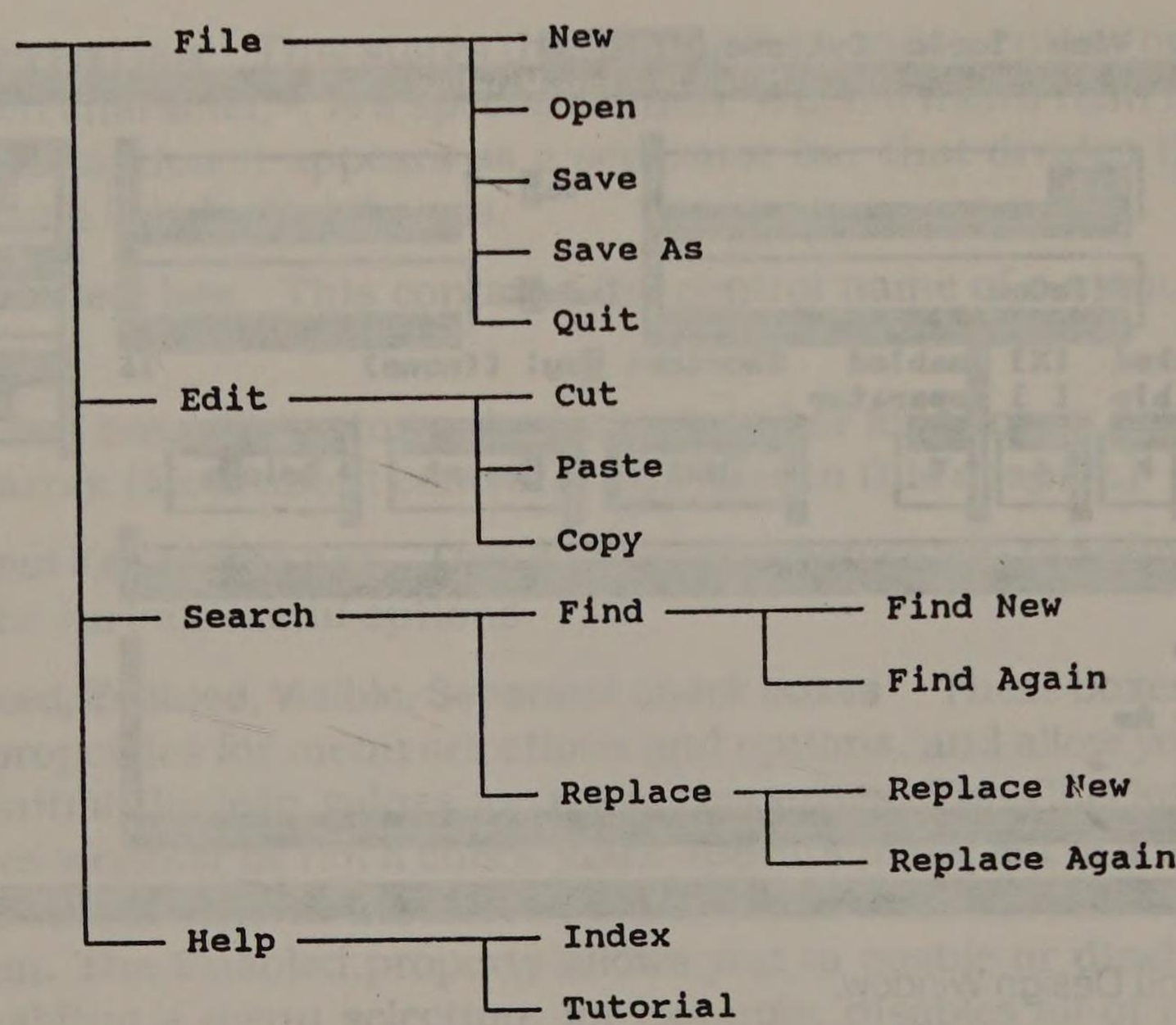
an outline in your word processor. In a text outline, you structure your thoughts by reducing the indentation level of more important items and increasing the indentation level of detailed items. The Visual Basic menus work the same way—menu selections are not indented, while menu options are. The menu options that are attached to menu selections are indented once by pressing the right arrow button (the left arrow button unindents the current entry). Visual Basic allows nested options of up to five levels of indentation. Each time you indent a menu option, four dots appear to its left.

Figure 8-2 shows the diagram of a sample menu structure that has menu options nested to two levels. Figure 8-2 shows the menu entries around the Search selection. Notice how the menu options are indented and observe the number of dots that lead the menu options. You can create the menu structure one level at a time or in the sequence that observes the levels of indentations. Either approach is fine.

Building a bare-bones menu

Let's build the menu structure of Figure 8-1. Load Visual Basic and work with the new project blank form. Move to the Form Design environment and select the Menu Design Window option from the Window menu. The sought window appears. Perform the following steps to create the menu selections:

1. Click the Caption text box and type File.
2. Press the TAB key to move to the CtlName text box. Type in FileCom.



8-2 The diagram for a sample menu.

3. Click the Next command button to move to the end of the menu outline.
4. Click the Insert command button to insert a new menu selection.
5. Repeat steps 1 to 4 for the Edit, Search, and Help menu selections. Use the EditCom, SearchCom, HelpCom names as the control names for their respective selections.

To create the menu options (that are indented once) repeat the following steps for each menu option:

1. Click on the menu selection or menu option that comes after the one you wish to insert.
2. Click on the Insert command button to insert a new menu item.
3. Click on the Caption text box and type in the name of the menu option. Use the names that appear in FIG. 8-1 as the captions.
4. Press the TAB key to move to the CtlName text box.
5. Type in the control name for the current menu item; append the letters Com to the caption name and remove any spaces within the caption name. For example, the caption Save As should have the control name SaveAsCom.
6. Press the right arrow button to indent the menu option. Nested menus require that you press the right arrow button twice. Press

the left arrow button to unindent any entry that you might erroneously over-indent.

Insert a hyphen menu item between the Save As and the Quit menu options to display a separator bar between these two entries. Use the Sep1 control name for the separator bar.

☞ The above steps are not the only way to build a menu structure. For example, you can insert a new menu item and then move it to the proper location.

Once you have finished designing the menu, click on the Done command button. Now you are back to the form. Notice that the menu selections appear on the form. Move to the Programming environment and select the QuitCom_Click event-handler. Insert an End statement inside the QuitCom_Click event-handling procedure. Run the program by pressing the F5 function key and experiment with the menu structure. Figure 8-3 shows one of the nested menu options. Clicking on any menu selection or option, except the Quit menu option, accomplishes nothing. When you are finished with the program, select the Quit option from the File menu.

Save the form and project files using the MTMENU.FRM and MTMENU.MAK filenames (invariably in directory \VBDOS\EASYVB).

Controlling menus at run time

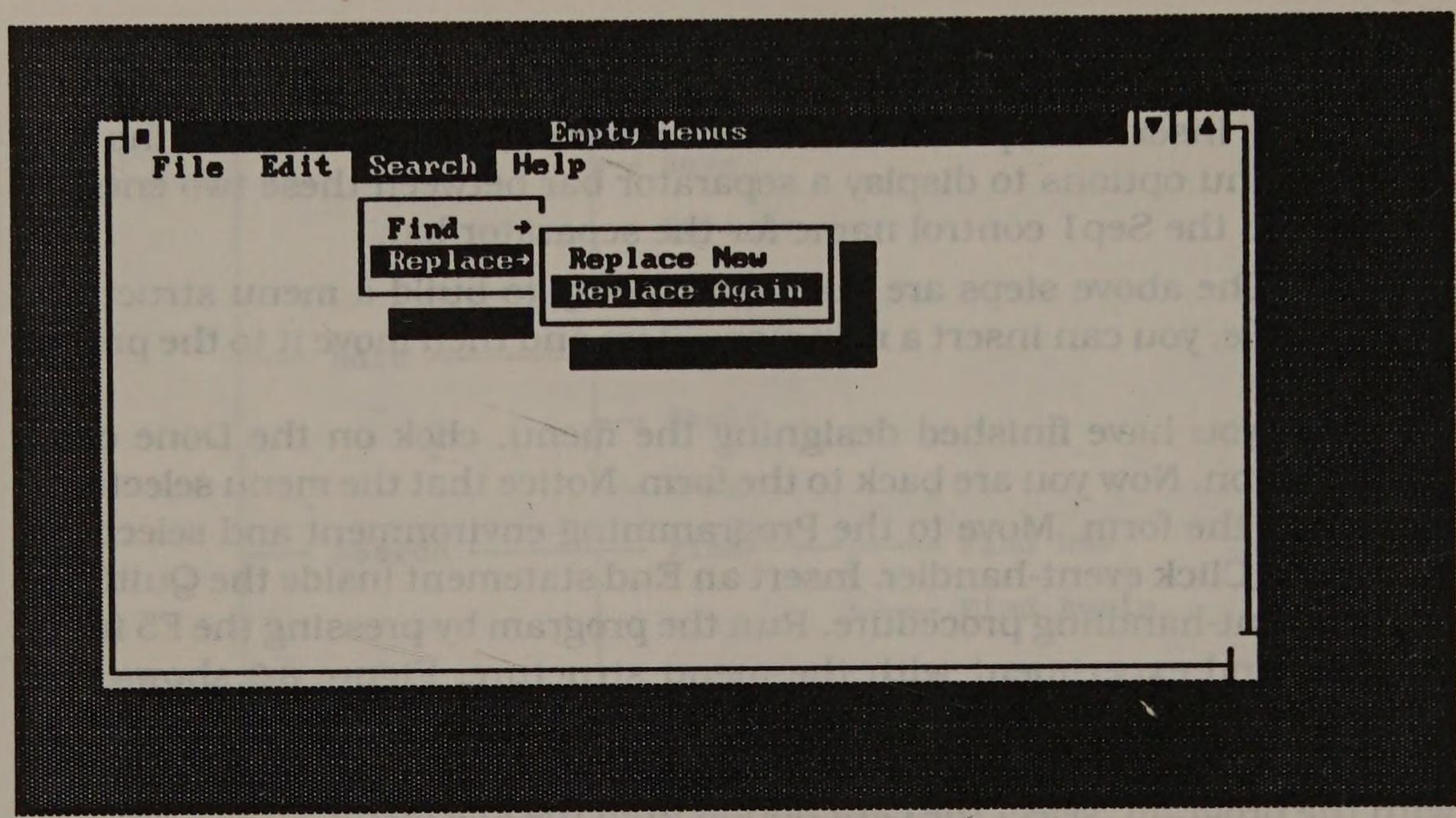
The Checked, Enabled, and Visible properties can be set and retrieved at run time in a manner very similar to the Toolbox control properties. The general syntax for accessing the menu properties is shown next:

```
menuItem.Checked [= boolean%]
menuItem.Enabled [= boolean%]
menuItem.Visible [= boolean%]
menuItem.Separator [= boolean%]
```

When the Checked property is assigned a TRUE (−1), a check mark appears before the caption of that menu item. The check mark is a visual flag that indicates the on/off state of a related condition.

Setting the Enabled and Visible property to FALSE (0) disables that menu item and all of the menu items to which it is connected. The effect on the Enabled property is that the caption of the menu item appears in faded fonts. By contrast, the effect on the Visible property is that the menu item disappears completely from the menu. Setting the Enabled and Visible properties to TRUE (−1) restores their menu items.

The Separator property specifies whether a menu item is a separator. When you set this property to TRUE, the menu item appears as a separator and the menu item does not receive any event. By contrast, when you set the Separator property to FALSE, the menu item appears as a normal item and responds to events unless its caption is a hyphen, which also makes it a separator.



8-3 A sample session with the MTMENU project, showing nested menus.

The menu-drive text file viewer

The text file viewer presented in the last chapter used command buttons to carry out the various tasks. This section presents a menu-driven version. But first, I want to draw your attention to FIG. 8-3, which shows a suggested documentation form for Visual Basic menus.

Figures 8-4 to 8-7 contain the documentation for the menu-driven text file viewer.

Listing 8-1 contains the code for the menu-drive file viewer. Attaching code to the menu items is easy and involves using the Object combo list box in the Code window. The event-handling procedures greatly resemble their counterparts in the FVIEW1 project. The main difference is that the command button names are replaced with menu item control names. Secondary program aspects to look at are:

The Toggle Read menu option Toggles the Enabled property of the Read menu option.

The Toggle OS menu option Toggles the Visible property of the DOS Command menu option, DOS/ Input text box, and the label for the latter box.

The Find menu option Uses its Checked property to inform you of the status of the last find. If the last text search was successful, the Checked property is set to TRUE to display the check mark.

If you change the text in the Find Input text box, the FindCom.Checked property is set to FALSE. The procedure FindBox_Change handles the event of changing text in the Find Input text box.

Run the program and experiment with the various menu options. No-

Application Name: Menu-Driven Text File Viewer	
Application Code Name: FVIEW2A	
Version: 1.0 Date Created: July 2, 1992	
Programmer(s): Namir Clement Shammas	
	List of Filenames
Storage Path:	C:\VBDOS\EASYVB
Project	FVIEW2A.MAK
Form 1	FVIEW2A.FRM

8-4 General project specifications for the FVIEW2A project.

Form # 1		Form Filename: FVIEW2A.FRM
Version: 1.0		Date: July 2, 1992
Control object type	**Default CtlName**	**Purpose**
Text Box	Text1	Contains the filename to read.
	Text2	Contains the text search string.
	Text3	Contains the DOS command line.
	Text4	Displays the text lines of a file.
Label	Label1	Label of Text1.
	Label2	Label of Text2.
	Label3	Label of Text3.

8-5 List of control objects for the FVIEW2A project.

tice that the options of a pull-down menu are available either by clicking on them or by pressing the first letter of the option. In the next section I will discuss how to control the keys that invoke the various menu options. Figure 8-8 shows a sample session with the menu-driven text file viewer.

Assigning shortcut and access keys to menus

The problem with programs like the above menu-driven file viewer is the lack of any quick keys to help you rapidly perform frequent tasks. Such quick keys are helpful when you become proficient in using certain programs and develop a reflex for performing a sequence of tasks.

Visual Basic allows menus to use both shortcut and access keys. Shortcut keys are CTRL key combinations (CTRL-A to CTRL-Z) that quickly execute specific menu options. Access keys are ALT keys that quickly choose specific menu selections.

☞ You can assign shortcut keys to menu options only (the indented menu items in the Menu Design Window). Use the Shortcut Key combo box

Application (Code) Name: FVIEW2A
Form # 1
Version: 1.0 Date: June 27, 1992

Original control name	Property	New setting
Form	Caption	File Viewer (ver. 2A)
Text1	CtlName	FilenameBox
	Text	(empty string)
Text2	CtlName	FindBox
	Text	(empty string)
Text3	CtlName	DosBox
	Text	(empty string)
Text4	CtlName	TextBox
	Text	(empty string)
	MultiLine	TRUE
	ScrollBars	3 - both
Label1	CtlName	FilenameLbl
	Caption	Filename Input
Label2	CtlName	FindLbl
	Caption	Find Input
Label3	CtlName	DosLbl
	Caption	DOS/Win Input

8-6 The modified control settings for the FVIEW2A project.

Form # 1 Form Filename: FVIEW2A.FRM
Version: 1.0 Date: July 2, 1992

Caption	Property	Setting	Purpose
File	CtlName	FileCom	
Read	CtlName	ReadCom	Reads a file in Text1.
	indented	once	
Toggle Read	CtlName	ToggleReadCom	Toggles Read menu option.
	indented	once	
Find File	CtlName	FindFileCom	Finds a file.
	indented	once	
—	CtlName	Sep1	
	indented	once	
Quit	CtlName	QuitCom	Exits the application.
	indented	once	
Search	CtlName	SearchCom	
Find	CtlName	FindCom	Finds text in Text1.
	indented	once	
Command	CtlName	CommandCom	
DOS Command	CtlName	DosCom	Executes a DOS command.
	indented	once	
Toggle OS	CtlName	ToggleOsCom	Shows/hides command options.
	indented	once	

8-7 The menu structure for FVIEW2A.

to select the CTRL key as the shortcut key for the current menu option. Shortcut keys must be unique. In fact, Visual Basic objects have duplicate shortcut keys when you press the Done command button. By default, a menu option has no shortcut key assigned to it. The shortcut key also appears with the menu option to remind you of that shortcut key.

Assigning access keys for menu selections is similar to assigning access keys to command buttons. Place the ampersand character, &, in front of the caption character you want to use with the ALT key at run time. The ampersand itself is not displayed as part of the caption. For example, if you insert the ampersand before the letter F in the File caption, the ALT-F key becomes the access key to the File menu selection. Consequently, pressing the ALT-F key combination at run time selects the File menu and pulls down its menu options.

You can also assign access keys to menu options. The technique is the same as with the menu selections; however, the effect and the usage are different. For example, if you make the caption of the Find File option the string Find Fil&e, the letter E becomes the access code for the Find File menu option; however, you cannot access that menu option directly by pressing ALT-E. You still have to choose the File selection. Once the File pull-down menu appears, you can press the key E to invoke the Find File menu option.

Figure 8-9 shows the updated specifications for the menu-driven text file viewer. The new settings include shortcut and access keys. The form and project files for this modification are FVIEW2B.FRM and FVIEW2B.MAK, respectively. The difference between the FVIEW2A and FVIEW2B is only in the

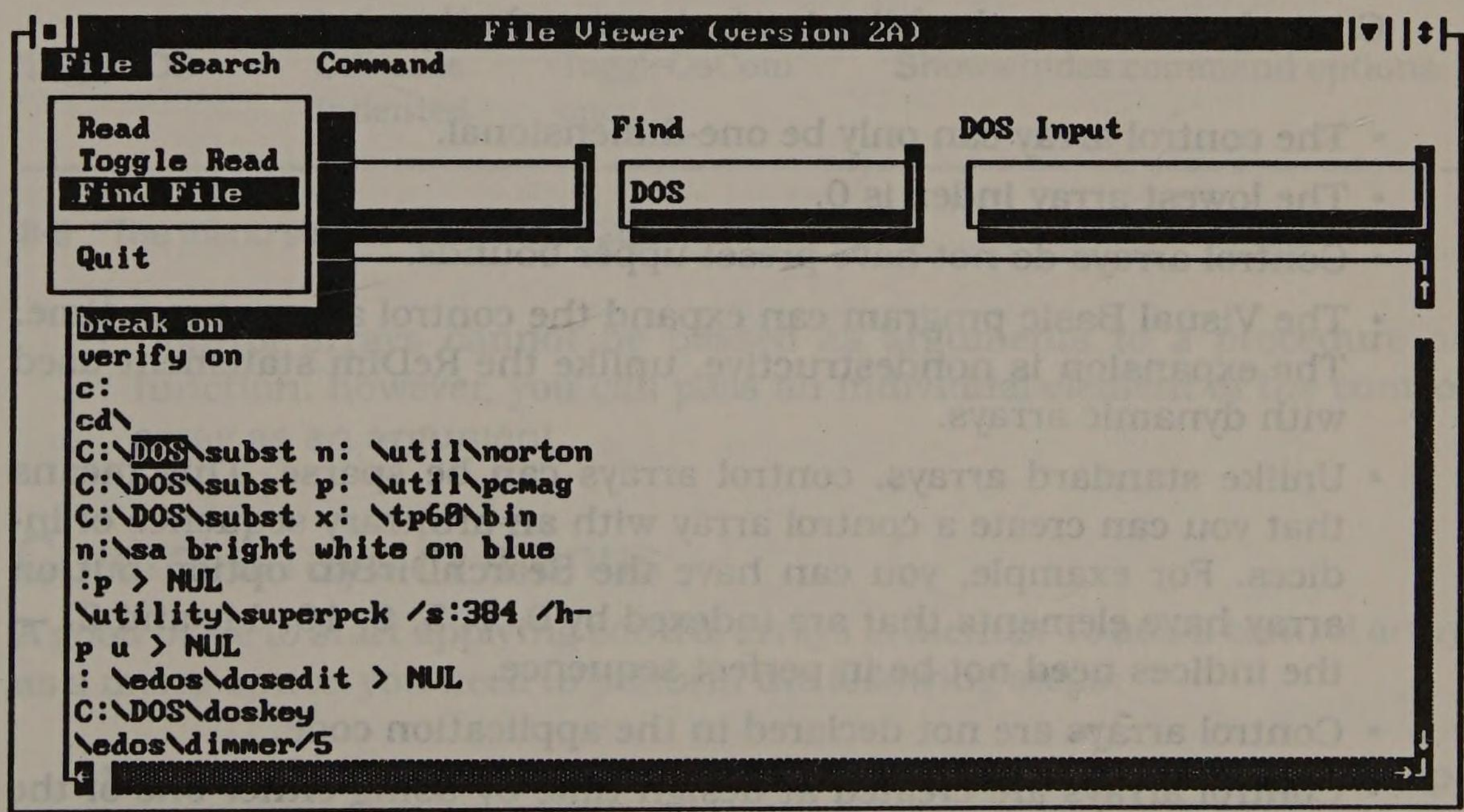
Listing 8-1 The code for the FVIEW2A project.

```
'$INCLUDE: 'CONSTANT.BI'


SUB DosCom_Click ()
        ' invoke the DOSBAT batch file
        SHELL ("dosbat.bat " + DosBox.Text)
END SUB

SUB FindBox_Change ()
    FindCom.CHECKED = FALSE
END SUB

SUB FindCom_Click ()
    STATIC CurIndex AS LONG
    DIM Find AS STRING
    ' update the index to the current character
    CurIndex = TextBox.SelStart + 1
    ' store the text of the Find box in the variable Find
    Find = FindBox.Text
    ' is variable Find empty?
    IF Find = "" THEN EXIT SUB ' nothing to find
```

Listing 8-1 Continued.

```
    ' locate the index of the next substring Find in the
    ' text box
    CurIndex = INSTR(CurIndex + 1, TextBox.Text, Find)
    ' found a match?
    IF CurIndex > 0 THEN
        ' Yes! Display the matching text as selected text
        TextBox.SelStart = CurIndex - 1
        TextBox.SelLength = LEN(Find)
        TextBox.SETFOCUS
        FindCom.CHECKED = TRUE
    ELSE
        ' No! Clear any selected text
        TextBox.SelStart = 0
        TextBox.SelLength = 0
        FindCom.CHECKED = FALSE
    END IF
END SUB

SUB FindFileCom_Click ()
    ' invoke the DOS dir command with the /s
    ' subdirectory option and the /p paginated
    ' output option
    SHELL ("dosbat.bat dir " + FilenameBox.Text + " /s/p")
END SUB

SUB QuitCom_Click ()
    END
END SUB

SUB ReadCom_Click ()
    DIM F AS STRING
    DIM L AS STRING
    DIM NL AS STRING * 2
    ' obtain filename from the filename text box
    F = FilenameBox.Text
    ' if filename is an empty string, exit
    IF F = "" THEN EXIT SUB
    NL = CHR$(13) + CHR$(10)
    ' set error-handler
    ON LOCAL ERROR GOTO BadFile
    ' open the file
    OPEN F FOR INPUT AS 1
    F = "" ' clear variable to reuse it
    ' loop to read the text lines from the ASCII file
    DO WHILE NOT EOF(1)
        LINE INPUT #1, L
        F = F + L + NL ' append a new line
    LOOP
    TextBox.Text = F ' copy F into text box
    ' close the file
```

```
    CLOSE #1
    ' exit procedure
    EXIT SUB
' ********** Error-handler *********
BadFile:
    BEEP
    MSGBOX "Cannot open file " + F, 0, "File I/O Error"
    RESUME EndOfSub
EndOfSub:
END SUB

SUB ToggleOSCom_Click ()
    ' toggle the Visible property of the DosBox text box,
    ' the DosBtn button, the WinBtn button, and the DosLbl
    ' label
    DosBox.Visible = NOT DosBox.Visible
    DosCom.Visible = NOT DosCom.Visible
    WinCom.Visible = NOT WinCom.Visible
    DosLbl.Visible = NOT DosLbl.Visible

END SUB

SUB ToggleReadCom_Click ()
    ReadCom.Enabled = NOT ReadCom.Enabled
END SUB
```



8-8 A sample session with project FVIEW2A.

menu design—the code is exactly the same. Make the required changes for the menu items (or load the project files). Run the modified menu-driven viewer and experiment with the various shortcut and access keys. This ver-

sion of the Visual Basic application begins to develop a more professional feel.

Control arrays

Visual Basic supports a new kind of array structure that is applied to controls. Control arrays share a number of properties with standard arrays (both fixed and dynamic), but they also have their differences.

Control arrays represent a group of controls that offer very similar functions. For example, consider an array of options boxes used in a text search dialog box. The control array is used to determine the search direction—forward, backward, or global.

Because control arrays contain similar individual controls, they can share the same event-handling procedures. For example, the array of the text-search option buttons can share the same SearchDirBtn_Click procedure, as shown below:

```
SUB SearchDirBtn_Click(Index AS INTEGER)
```

The Index parameter is required to select the proper control element in the array. Inside the event-handling procedure you can access the sought control using the index parameter. Thus, the controlArrayName(Index) terms accessed the Index'th control for the SearchDirBtn(Index) procedure.

Control arrays have the following features and rules:

- The control array can only be one-dimensional.
- The lowest array index is 0.
- Control arrays do not have preset upper bounds.
- The Visual Basic program can expand the control array at run time. The expansion is nondestructive, unlike the ReDim statement used with dynamic arrays.
- Unlike standard arrays, control arrays can be sparse. This means that you can create a control array with an arbitrary sequence of indices. For example, you can have the SearchDirBtn option button array have elements that are indexed by 0, 1, 5, 9, 10, 11, and 21—the indices need not be in perfect sequence.
- Control arrays are not declared in the application code.
- Control arrays are created at design time by using either one of the two following methods: in the first method, you assign the same name to the CtlName property of two similar controls. The second method creates a control array by assigning a value to the Index property (which is blank for controls that are members of a control array).
- You can change the setting of the Index property of a control at design time only.

Form # 1 Form Filename: FVIEW2B.FRM
Version: 1.0 Date: July 2, 1992

Caption	Property	Setting	Purpose
&File	CtlName	FileCom	
Read	CtlName	ReadCom	Reads a file in Text1.
	Shortcut	CTRL-R	
	indented	once	
Toggle Read	CtlName	ToggleReadCom	Toggles Read menu option.
	indented	once	
Find File	CtlName	FindFileCom	Finds a file.
	Shortcut	CTRL-F	
	indented	once	
—	CtlName	Sep1	
	indented	once	
Quit	CtlName	QuitCom	Exits the application.
	Shortcut	CTRL-Q	
	indented	once	
&Search	CtlName	SearchCom	
Find	CtlName	FindCom	Finds text in Text1.
	Shortcut	CTRL-I	
	indented	once	
&Command	CtlName	CommandCom	
DOS Command	CtlName	DosCom	Executes a DOS command.
	Shortcut	CTRL-D	
	indented	once	
Toggle OS	CtlName	ToggleOsCom	Shows/hides command options.
	indented	once	

8-9 The menu structure for FVIEW2B.

- Control arrays cannot be passed as arguments to a procedure or function; however, you can pass an individual element of the control array as an argument.

Control arrays in menus

A good place to start applying control arrays is menus. To add a control array as a menu option you need to perform the following steps:

1. Define the first element of the control array menus as a menu option. That menu option should have the following special settings:
   ~ The caption is an empty string.
   ~ The Visible property should be turned off.
   ~ The Index check box should have a 0.

   The menu option should be properly indented and have a control name. Visual Basic won't allow blank control names.

2. Create accompanying menu options that add and possibly delete control array menus. These menu options are declared just like any ordinary menu options and do not have any special settings.
3. Declare a variable in the general declarations section to keep track of the current number of additional control array menus. The menu counter variable should be assigned a 0 in the Form_Load procedure.
4. Insert Basic statements in the procedure that handles adding control array menus. The code for this procedure should perform the following steps:
   ~ Obtains the caption of the new menu option.
   ~ Increments the menu counter variable.
   ~ Loads a new menu option using the Load command. The general syntax for this command is:

     ```
     LOAD ctlName(menuCounter)
     ```

   ~ Assigns the new menu option its caption.
   ~ Sets the Visible property of the new menu option to TRUE.
5. Insert Basic statements in the procedure that handles deleting control array menus. The code for this procedure should perform the following steps:
   ~ Obtains the index of the menu option you want to delete.
   ~ Validates the obtain index.
   ~ Reassigns the captions of the menu options to override the delete one.
   ~ Unloads the targeted menu option using the Unload command. The general syntax for this command is:

     ```
     UNLOAD ctlName(menuCounter)
     ```

   ~ Decrements the menu counter variable.
6. Insert Basic code in the event-handling procedure for the control array menus. That procedure must have an integer index parameter, such as Index As Integer. The selection control array menu option is accessed using ctlName(Index).Caption.

Control array menus for the file viewer

The menu-driven file viewer can use control array menus to maintain the names of important files that you want to view (at least twice) during a program session. During run time you can specify the names of these important files and make them menu options. From then on, selecting these menu options automatically displays these files.

Following the steps mentioned in the last section, begin by updating the menu structure that appears in the Menu Design Window. Figure 8-10

shows the new structure of the new menu and reveals the following new menu items:

The Add File menu option Adds filenames as new control array menu options.

The Delete File menu option Deletes an existing control array menu option.

The first control array menu option With the control name of QuickFilename and the Index of 0.

Additional hyphen menu options Divide the File pull-down menu into four logical groups.

Listing 8-3 shows the code for the new version of the file viewer (dubbed the FVIEW2C project). The general declaration section contains a declaration for the Integer-typed variable, NumFiles, that maintains the current number of additional control array menu options. Notice that the Form_Load procedure initializes NumFiles to 0. While this assignment is optional, because Visual Basic initializes NumFiles to 0 anyway, it is a safer programming practice that produces clearer code.

Listing 8-2 contains the following procedures that manage the control array menus:

The AddFileCom_Clock procedure Handles adding new control array menu options:

```
SUB AddFileCom_Click ()
    DIM TheFilename AS STRING
    TheFilename = INPUTBOX$("Enter full file name: ")
    NumFiles = NumFiles + 1 ' increment the menu count
    LOAD QuickFilename(NumFiles) ' load a new menu command
    QuickFilename(NumFiles).Caption = TheFilename
    QuickFilename(NumFiles).Visible = TRUE
END SUB
```

The procedure performs the following tasks:

- Uses the InputBox$ function to prompt the application user for a filename (stored in the variable TheFilename).
- Increments the variable NumFiles.
- Loads a new control array menu and associates the NumFiles index with that array member.
- Assigns the variable The Filename to the caption of the new control array menu option.
- Sets the Visible property of the new array member to TRUE.

The DeleteFileCom_Click procedure Handles removing control array menu options:

Form # 1 Form Filename: FVIEW2C.FRM
Version: 1.0 Date: July 2, 1992

Caption	Property	Setting	Purpose
&File	CtlName	FileCom	
Read	CtlName	ReadCom	Reads a file in Text1.
	Shortcut	CTRL-R	
	indented	once	
Toggle Read	CtlName	ToggleReadCom	Toggles Read menu option.
	indented	once	
Find File	CtlName	FindFileCom	Finds a file.
	Shortcut	CTRL-F	
—	indented	once	
	CtlName	Sep1	
	indented	once	
Add File	CtlName	AddFileCom	Adds a filename.
	indented	once	
Delete File	CtlName	DeleteFileCom	Deletes a filename.
	indented	once	
—	CtlName	Sep2	
	indented	once	
(noname)	CtlName	QuickFilename	
	Index	0	
	Visible	off	
	indented	once	
—	CtlName	Sep3	
	indented	once	
Quit	CtlName	QuitCom	Exits the application.
	Shortcut	CTRL-Q	
	indented	once	
&Search	CtlName	SearchCom	
Find	CtlName	FindCom	Finds text in Text1.
	Shortcut	CTRL-Q	
	indented	once	
&Command	CtlName	CommandCom	
DOS Command	CtlName	DosCom	Executes a DOS command.
	Shortcut	CTRL-D	
	indented	once	
Toggle OS	CtlName	ToggleOsCom	Shows/hides command options.
	indented	once	

8-10 The menu structure for the FVIEW2C project.

Listing 8-2 The code for the FVIEW2C project.

```
DECLARE SUB ReadCom_Click ()
'$INCLUDE: 'CONSTANT.BI'
DIM SHARED NumFiles AS INTEGER
```

```
SUB AddFileCom_Click ()
    DIM TheFilename AS STRING
    TheFilename = INPUTBOX$("Enter full file name: ")
    NumFiles = NumFiles + 1 ' increment the menu count
    LOAD QuickFilename(NumFiles) ' load a new menu command
    QuickFilename(NumFiles).Caption = TheFilename
    QuickFilename(NumFiles).Visible = TRUE
END SUB

SUB DeleteFileCom_Click ()
    DIM N AS INTEGER: DIM I  AS INTEGER
    N = VAL(INPUTBOX$("Enter number to delete: "))
    IF (N > 0) AND (N <= NumFiles) THEN
        FOR I = N TO NumFiles - 1
            QuickFilename(I).Caption = QuickFilename(I + 1).Caption
        NEXT I
        UNLOAD QuickFilename(NumFiles)
        NumFiles = NumFiles - 1
    ELSE
        MSGBOX "The number is out-of-range", 0, "Input Error"
    END IF
END SUB

SUB DosCom_Click ()
        ' invoke the DOSBAT batch file
        SHELL ("dosbat.bat " + DosBox.Text)
END SUB

SUB FindBox_Change ()
    FindCom.CHECKED = 0
END SUB

SUB FindCom_Click ()
    STATIC CurIndex AS LONG
    DIM Find AS STRING
    ' update the index to the current character
    CurIndex = TextBox.SelStart + 1
    ' store the text of the Find box in the variable Find
    Find = FindBox.Text
    ' is variable Find empty?
    IF Find = "" THEN EXIT SUB ' nothing to find
    ' locate the index of the next substring Find in the
    ' text box
    CurIndex = INSTR(CurIndex + 1, TextBox.Text, Find)
    ' found a match?
    IF CurIndex > 0 THEN
        ' Yes! Display the matching text as selected text
        TextBox.SelStart = CurIndex - 1
        TextBox.SelLength = LEN(Find)
        TextBox.SETFOCUS
        FindCom.CHECKED = TRUE
```

Listing 8-2 Continued.

```
    ELSE
        ' No! Clear any selected text
        TextBox.SelStart = 0
        TextBox.SelLength = 0
        FindCom.CHECKED = FALSE
    END IF
END SUB

SUB FindFileCom_Click ()
    ' invoke the DOS dir command with the /s
    ' subdirectory option and the /p paginated
    ' output option
    SHELL ("dosbat.bat dir " + FilenameBox.Text + " /s/p")
END SUB

SUB FindLbl_Change ()
    FindCom.CHECKED = FALSE
END SUB

SUB Form_Load ()
    NumFiles = 0
END SUB

SUB QuickFilename_Click (Index AS INTEGER)
    FilenameBox.Text = QuickFilename(Index).Caption
    ReadCom_Click
END SUB

SUB QuitCom_Click ()
    END
END SUB

SUB ReadCom_Click ()
    DIM F AS STRING
    DIM L AS STRING
    DIM NL AS STRING * 2
    ' obtain filename from the filename text box
    F = FilenameBox.Text
    ' if filename is an empty string, exit
    IF F = "" THEN EXIT SUB
    NL = CHR$(13) + CHR$(10)
    ' set error-handler
    ON LOCAL ERROR GOTO BadFile
    ' open the file
    OPEN F FOR INPUT AS 1
    F = "" ' clear variable to reuse it
    ' loop to read the text lines from the ASCII file
    DO WHILE NOT EOF(1)
        LINE INPUT #1, L
        F = F + L + NL ' append a new line
```

```
    LOOP
    TextBox.Text = F ' copy F into text box
    ' close the file
    CLOSE #1
    ' exit procedure
    EXIT SUB
' ********** Error-handler *********
BadFile:
    BEEP
    MSGBOX "Cannot open file " + F, 0, "File I/O Error"
    RESUME EndOfSub
EndOfSub:
END SUB

SUB ToggleOSCom_Click ()
    ' toggle the Visible property of the DosBox text box,
    ' the DosBtn button, the WinBtn button, and the DosLbl
    ' label
    DosBox.Visible = NOT DosBox.Visible
    DosCom.Visible = NOT DosCom.Visible
    WinCom.Visible = NOT WinCom.Visible
    DosLbl.Visible = NOT DosLbl.Visible

END SUB

SUB ToggleReadCom_Click ()
    ReadCom.Enabled = NOT ReadCom.Enabled
END SUB
```

```
SUB DeleteFileCom_Click ()
    DIM N AS INTEGER: DIM I AS INTEGER
    N = VAL(INPUTBOX$("Enter number to delete: "))
    IF (N > 0) AND (N = NumFiles) THEN
        FOR I = N TO NumFiles - 1
            QuickFilename(I).Caption = QuickFilename(I +
            1).Caption
        NEXT I
        UNLOAD QuickFilename(NumFiles)
        NumFiles = NumFiles - 1
    ELSE
        MSGBOX "The number is out-of-range", 0, "Input Error"
    END IF
```

The procedure performs the subsequent tasks:

- Prompts the application user to enter the index of the menu option to delete.
- If the input is within valid range reassign the captions, unload the target menu option, and decrease the NumFiles variable.

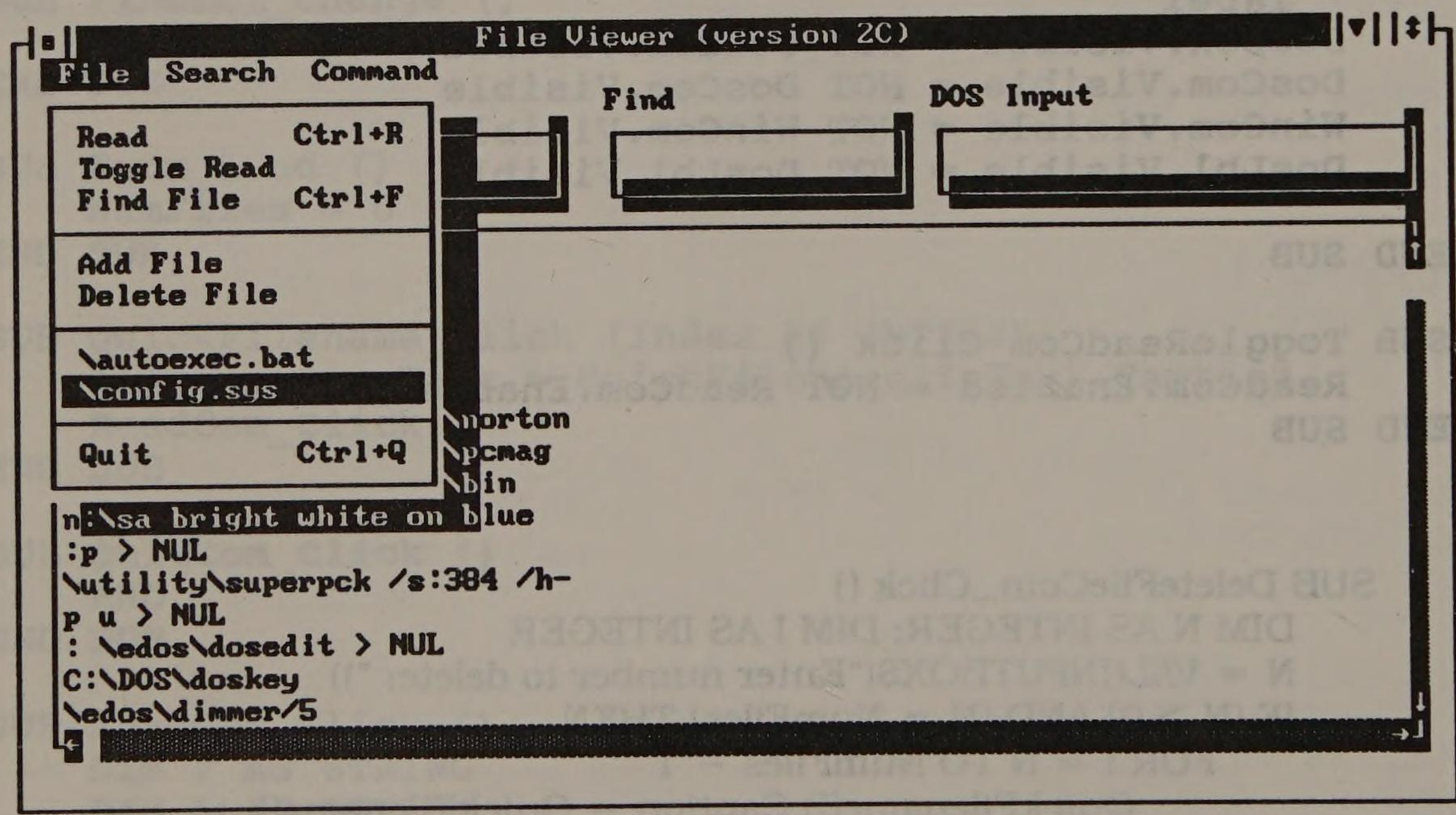
- If the input is outside the valid range, display an error message box.

The QuickFilename_Click procedure Quickly loads the selected filename:

```
SUB QuickFilename_Click (Index AS INTEGER)
    FilenameBox.Text = QuickFilename(Index).Caption
    ReadCom_Click
END SUB
```

The procedure copies the caption of the selected control array menu option into the Text property of the Filename Input text box, then calls the ReadCom_Click procedure. Notice that the expression QuickFIlename(Index).Caption is used to access the sought menu option.

Run the FVIEW2C program and use the Add File menu options to add a few text files. Once you build this list, select these menu options to quickly load the text files.



8-11 A sample session with project FVIEW2C.

9
Working with multiple forms

The Visual Basic examples presented so far use a single form. As you become more proficient in programming with Visual Basic, you will find that a single form is limiting—there are so many controls that you can draw on a form. This chapter looks at using multiple forms in a Visual Basic application.

Using multiple forms performs two essential goals: it gives you more form space, and allows you to create forms that logically group a family of controls to perform a specific task. In a manner of speaking, the additional forms are to an application what a SUB procedure is to Visual Basic.

Visual Basic offers a number of *common* dialog boxes that you can find in the CMNDLG.BAS module and the CMNDLGF.FRM form. These dialog boxes are the File Open, File Save, File Print, Find Text, Change Text, Color Palette, and About dialog boxes.

Dialog boxes and modal forms

A Visual Basic application that uses multiple forms designates a primary form. This is the form that is automatically loaded into memory and displayed. All other forms are secondary and must be explicitly loaded into memory and displayed. These secondary forms can be modal. A modal form is one that does not allow you to switch focus to another window until that form is closed. Dialog boxes are invariably modal windows, because they require you to input data before moving on to other forms. Typically, modal windows cannot be minimized or maximized and have a border style that is different from modeless windows.

Methods and statements that manage forms

New forms are created using the New Form option in the File menu. Visual Basic offers the following two methods and two statements to manage the forms at run time:

- The LOAD statement loads a form into the working memory. This step enables other in-memory forms to access the controls and properties of that form; however, the LOAD statement does not automatically show the form. The general syntax for the LOAD statement is:

  LOAD formName

- The UNLOAD statement removes a form from memory and from the screen and loses the data in that form. The controls and properties of the removed form are no longer accessible to other in-memory forms. The general syntax for the UNLOAD statement is:

  UNLOAD formName

Programming tip Accumulating forms in memory bleeds the memory. Unload infrequently used forms when you have finished with them. Keep only those forms that are frequently used to maintain good program speed.

- The SHOW method first loads a form (if it is not already in memory) and then displays it. If the specified modal state is 1, the loaded form is modal. A 0 modal state (the default if the modal argument is omitted) makes the form modeless. The general syntax for the Show method is:

  [form.]SHOW [modalState%]

Programming tip When a form is loaded by either the LOAD statement or the SHOW method, the Form_Load procedure of that method is automatically invoked. This feature enables you to include additional or special form initialization in the Form_Load procedure.

- The HIDE method makes the form invisible (sets the Visible property of that form to FALSE) while maintaining its data. Other in-memory forms can still access the controls and properties of a hidden form. The general syntax for the HIDE method is:

  [form.]HIDE

The general syntax for accessing a property in another form is:

formName.propertyName

The general syntax for accessing the property of a control in another form is:

formName.controlName.propertyName

☞ If you try to access the property or controls of a form that is currently not in memory, the run-time system will automatically load that form. This interesting feature absolves you from having to set error trapping procedures to protect against accessing a form that is not loaded in memory.

Setting the form properties

The preceding discussions made an important distinction between modal and modeless forms (and especially the windows they produce at run time). Moreover, modal forms were associated with dialog boxes. This section discusses the form properties that fine tune the visual appearance and behavior of general forms and dialog boxes. During run time general forms can be resized, minimized, and maximized, and might give the focus to other windows. By contrast, dialog boxes are more self-centered, so to speak. You can only move a typical dialog box. You are not allowed to resize, minimize, maximize, or switch to another window until you are finished with the dialog box.

Table 9-1 lists the properties for the general forms and the dialog boxes that are relevant in their display and behavior at run time. The table also includes the recommended settings, indicating which settings are default ones. The settings for the general forms and the dialog boxes reflect the typical behavior previously described.

Selecting the startup form or module

A quick quiz question: where does a multiform Visual Basic program start executing? If your answer is "in the main form" you are right! The main form is the default form that is automatically loaded and executed. However, Visual Basic offers two other routes for starting up a multiform Visual Basic program:

- Selecting a secondary form as the startup form. This selection enables a Visual Basic to start running a secondary form instead of the main form. Such a selection is made possible by choosing the Set Startup Form option from the Run menu. Visual Basic displays a dialog box with a list of current forms to choose from.
- Selecting a startup module. Rather than choosing any one particular form as the startup point, you can begin execution in a module. The rule to follow is that the startup module must contain a procedure called Main. This procedure does not load any form.

Table 9-1 The properties of general forms and dialog boxes.

Form property	Setting	Comments
BorderStyle	0	No caption, no Minimize or Maximize buttons, no Control-menu, no title bar, and not sizable.
	1	Fixed single-line border. Sizable only with Minimize and Maximize buttons. Optional Control-menu box, title bar, Minimize and Maximize buttons.
	2	(Default.) Sizable single-line border. The border can be resized with any one of the above controls.
	3	Fixed double-line border. Not sizable and no Minimize or Maximize buttons. Optional Control-menu box and title bar.
	4	Sizable double-line border.
	5	Fixed solid.
	6	Sizable solid.
MinButton	0	No Minimize button.
	−1	(Default.) Minimize button.
MaxButton	0	No Maximize button.
	−1	(Default) Maximize button.
ControlBox	−1	(Default.) Displays the Control-menu box.
	0	Removes the Control-menu box.
Caption	String	The text for the caption.
FormName	String	The name that identifies the form. Not available at run time.

☞ Keep in mind the following for loading forms. The default or your selected startup form are automatically loaded and displayed. All other forms must be explicitly loaded and displayed when needed. If you start up with a Basic module, then you must explicitly load all of the forms (when each is needed).

An example

Having discussed the various aspects of a multiform Visual Basic application, let's see an example. The program presented in this section basically converts among lengths, temperatures, and numeric bases. The program consists of a main control center form (the main form) that contains buttons that invoke the secondary forms. Each of the three secondary forms is a simple dialog box that converts among the following:

- Meters and feet.
- Celsius and Fahrenheit temperatures.
- Decimal, hexadecimal, and octal numbers.

The main form acts as a focal point. You invoke a conversion form, perform the related calculations, and then return to the focal point.

Figures 9-1 though 9-9 describe the specifications for the four forms involved. The form files and control names are FORMS1.FRM (Form), LENGTHFO.FRM (LengthForm), TEMPFORM.FRM (TempForm), and BASEFORM.FRM (BaseForm). The specifications indicate the custom form settings needed to make dialog boxes out of the secondary forms (in addition to specifying a modal state of 1 when showing these forms). The project filename is FORMS1.MAK. Figures 9-10 through 9-13 show screen images of the four forms.

Application Name: Conversions Application
Application Code Name: FORMS1
Version: 1.0 Date Created: July 3, 1992
Programmer(s): Namir Clement Shammas

	List of filenames
Storage Path:	\VBDOS\EASYVB
Project	FORMS1.MAK
Form 1	FORMS1.FRM
Form 2	LENGTHFO.FRM
Form 3	TEMPFORM.FRM
Form 4	BASEFORM.FRM

9-1 The general specifications for the FORMS1.MAK project.

Form # 1 Form Filename: FORMS1.FRM
Version: 1.0 Date: July 3, 1992

Control object type	**Default CtlName**	**Purpose**
Command button	Command1	Invokes the LengthForm.
	Command2	Invokes the TempForm.
	Command3	Invokes the BaseForm.
	Command4	Exits the application.

9-2 Basic specifications for the main form.

Listings 9-1 through 9-4 contain the attached code for each form. Notice that the general declarations section of the main form contains several $FORM metacommands. These metacommands are needed to allow the access of forms properties by other forms. The procedures handling the Click event for the command buttons in the main form invoke the Show method on the different secondary forms. In each case, a modal of 1 is specified. This value, combined with the custom form settings, produces dialog boxes. Also notice that the QuitBtn_Click procedure contains the End statement that is needed to exit this multiform application.

The code attached to secondary forms contains the event-handling procedures for the text boxes and the return-to-main command button. The

Application (Code) Name: FORMS1
Form # 1
Version: 1.0 Date: July 3, 1992

Original control name	Property	New setting
Form	Caption	Control Center
Command1	CtlName	LengthBtn
	Caption	Convert Length
Command2	CtlName	TempBtn
	Caption	Convert Temp.
Command3	CtlName	BaseBtn
	Caption	Convert Base
Command4	CtlName	QuitBtn
	Caption	Quit

9-3 Updated settings for the main form.

Form # 2 Form Filename: LENGTHFO.FRM
Version: 1.0 Date: July 3, 1992

Control object type	Default CtlName	Purpose
Text Box	Text1	Stores meters.
	Text2	Stores inches.
Label	Label1	Label for meter box.
	Label2	Label for inch box.
Command Button	Command1	Returns to the main form.

9-4 Basic specifications for the LENGTHFO.FRM form.

text boxes trap the LostFocus event, which is generated when you TAB out of that text box or click on another control. This event signals to the event-handler that you have finished keying in a new number or editing the one currently in the selected text box. Consequently, the numbers in the other text boxes (in the same form) are also updated. Using this technique enables the program to work without additional command buttons that trigger the numeric conversions.

The ReturnBtn_Click event-handler implemented in the secondary forms unloads its respective form. This action results in hiding the form, moving it from memory, and returning to the main form.

Run the program and select the length conversion, for example. Select the Meters text box and type 1. Press the TAB key to switch to another control, and watch the equivalent of 1 meter in feet appear in the Feet text box. Select the Feet text box and overwrite the current contents with 10. Press

Application (Code) Name: FORMS1
Form # 2
Version: 1.0 Date: July 3, 1992

Original control name	Property	New setting
Form	Caption	Length Conversion
	BorderStyle	3 - Fixed Double
	MinButton	FALSE
	MaxButton	FALSE
	FormName	LengthForm
Text1	CtlName	MeterBox
	Text	(Empty string.)
Text2	CtlName	FeetBox
	Text	(Empty string.)
Label1	CtlName	MeterLbl
	Caption	Meters
Label2	CtlName	FeetLbl
	Caption	Feet
Command1	CtlName	ReturnBtn
	Caption	Return To Main

9-5 Updated settings for the LENGTHFO.FRM form.

Form # 3 Form Filename: TEMPFORM.FRM
Version: 1.0 Date: July 3, 1992

Control object type	Default CtlName	Purpose
Text Box	Text1	Stores Celsius degrees.
	Text2	Stores Fahrenheit degrees.
Label	Label1	Label for Celsius box.
	Label2	Label for Fahrenheit box.
Command Button	Command1	Returns to the main form.

9-6 Basic specifications for the TEMPFORM.FRM form.

the TAB key and watch the equivalent of 10 feet appear in the Meters text box. Click on the Return To Main command box to go back to the main form. Experiment further with the program by selecting the other conversions.

As an exercise, change the Unload statements in the ReturnBtn_Click procedures into formName.Hide. Run the program and use any one of the conversion forms twice. What do you see? The results from the previous visit are still there, because the form was hidden.

Application (Code) Name: FORMS1
Form # 3
Version: 1.0 Date: July 3, 1992

Original control name	Property	New setting
Form	Caption	Temperature conversion.
	BorderStyle	3 - Fixed double
	MinButton	FALSE
	MaxButton	FALSE
	FormName	TempForm
Text1	CtlName	CBox
	Text	(Empty string.)
Text2	CtlName	FBox
	Text	(Empty string.)
Label1	CtlName	CLbl
	Caption	Celsius
Label2	CtlName	FLbl
	Caption	Fahrenheit
Command1	CtlName	ReturnBtn
	Caption	Return To Main

9-7 Updated settings for the TEMPFORM.FRM form.

Form # 4 Form Filename: BASEFORM.FRM
Version: 1.0 Date: July 3, 1992

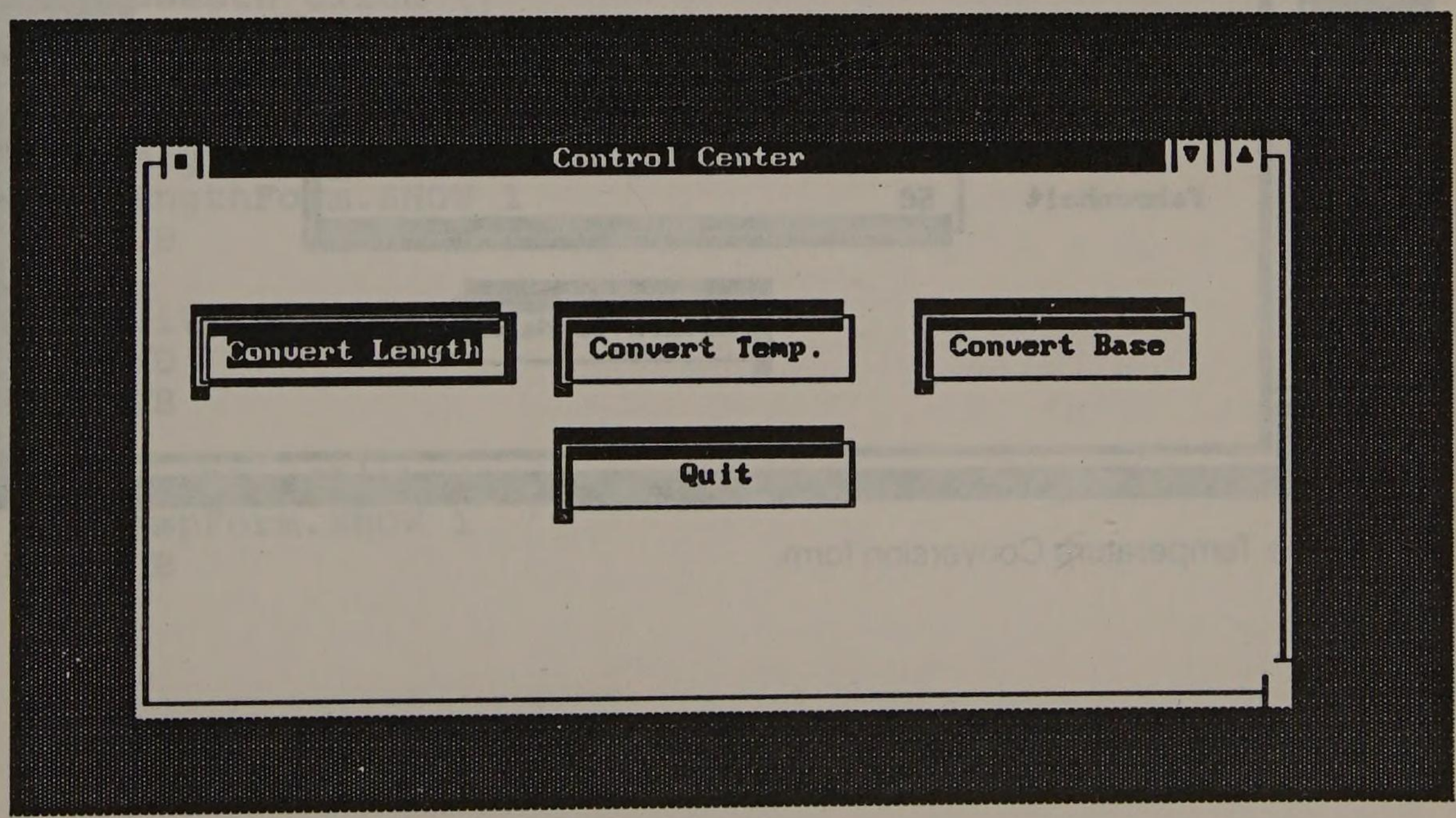
Control object type	Default CtlName	Purpose
Text Box	Text1	Stores a decimal number.
	Text2	Stores a hexadecimal number.
	Text3	Stores an octal number.
Label	Label1	Label for the decimal number.
	Label2	Label for the hexadecimal number.
	Label3	Label for the octal number.
Command Button	Command1	Returns to the main form.

9-8 Basic specifications for the BASEFORM.FRM form.

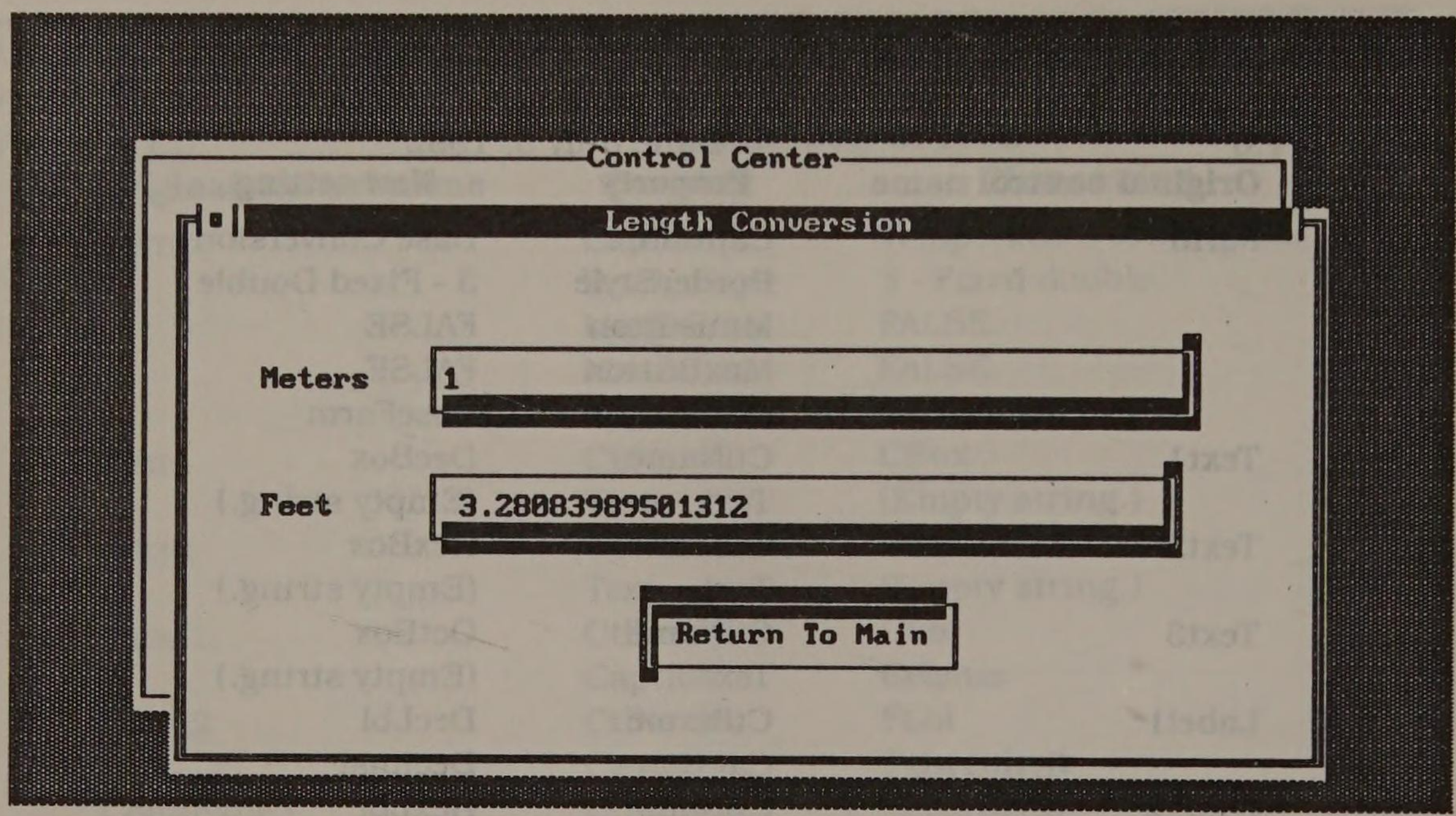
Application (Code) Name: FORMS1
Form # 4
Version: 1.0 Date: July 3, 1992

Original control name	Property	New setting
Form	Caption	Base Conversion
	BorderStyle	3 - Fixed Double
	MinButton	FALSE
	MaxButton	FALSE
	FormName	BaseForm
Text1	CtlName	DecBox
	Text	(Empty string.)
Text2	CtlName	HexBox
	Text	(Empty string.)
Text3	CtlName	OctBox
	Text	(Empty string.)
Label1	CtlName	DecLbl
	Caption	Decimal
Label2	CtlName	HexLbl
	Caption	Hexadecimal
Label3	CtlName	OctLbl
	Caption	Octal
Command1	CtlName	ReturnBtn
	Caption	Return To Main

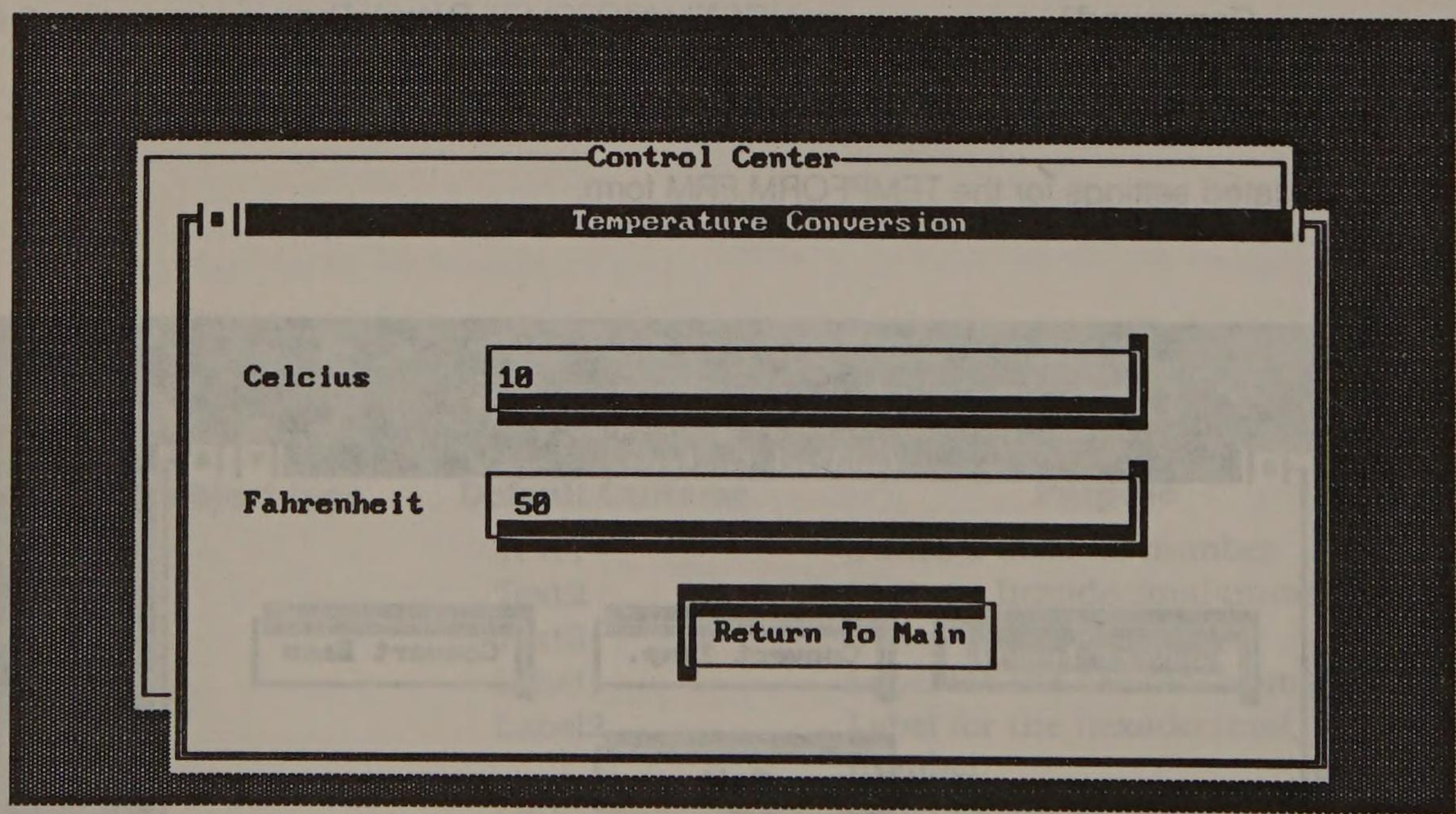
9-9 Updated settings for the TEMPFORM.FRM form.



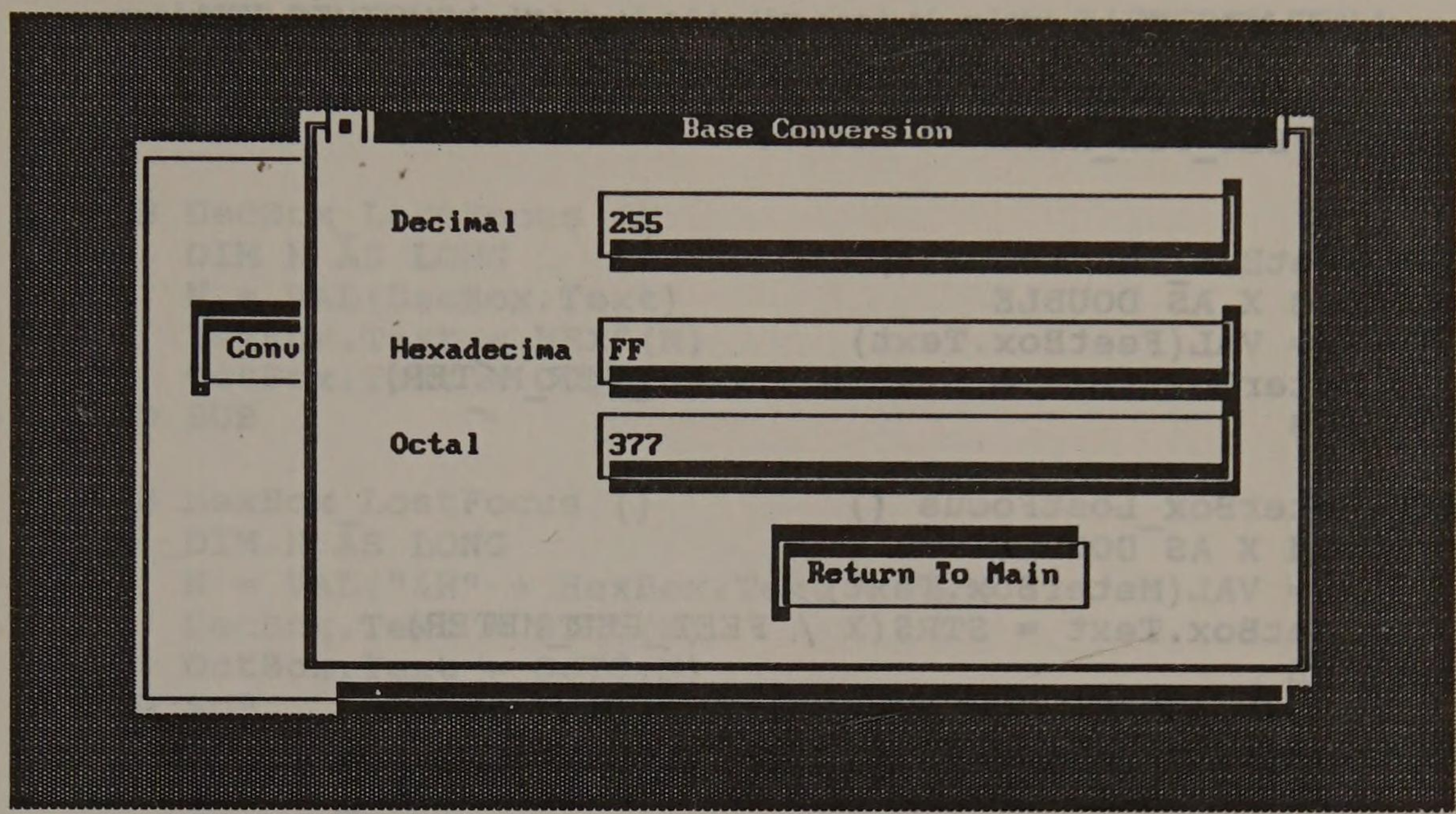
9-10 The Control Center form.



9-11 The Length Conversion form.



9-12 The Temperature Conversion form.



9-13 The Base Conversion form.

Listing 9-1. The code attached to the controls of the main form.

```
'$FORM BaseForm
'$FORM TempForm
'$FORM LengthForm
'$INCLUDE: 'CONSTANT.BI'

SUB BaseBtn_Click ()
    BaseForm.SHOW 1
END SUB

SUB LengthBtn_Click ()
    LengthForm.SHOW 1
END SUB

SUB QuitBtn_Click ()
    END
END SUB

SUB TempBtn_Click ()
    TempForm.SHOW 1
END SUB
```

Listing 9-2 The code attached to the controls of the LENGTHFO.FRM form.

```
'$INCLUDE: 'CONSTANT.BI'
CONST FEET_PER_METER = .3048#


SUB FeetBox_LostFocus ()
    DIM X AS DOUBLE
    X = VAL(FeetBox.Text)
    MeterBox.Text = STR$(X * FEET_PER_METER)
END SUB

SUB MeterBox_LostFocus ()
    DIM X AS DOUBLE
    X = VAL(MeterBox.Text)
    FeetBox.Text = STR$(X / FEET_PER_METER)
END SUB

SUB ReturnBtn_Click ()
    UNLOAD LengthForm
END SUB
```

Listing 9-3 The code attached to the controls of the TEMPFORM.FRM form.

```
'$INCLUDE: 'CONSTANT.BI'


SUB CBox_LostFocus ()
    DIM X AS DOUBLE
    X = VAL(CBox.Text)
    FBox.Text = STR$(32# + 1.8# * X)
END SUB

SUB FBox_LostFocus ()
    DIM X AS DOUBLE
    X = VAL(FBox.Text)
    CBox.Text = STR$((X - 32#) / 1.8#)
END SUB

SUB ReturnBtn_Click ()
    UNLOAD TempForm
END SUB
```

Listing 9-4 The code attached to the controls of the BASEFORM.FRM form.

```
'$INCLUDE: 'CONSTANT.BI'

SUB DecBox_LostFocus ()
    DIM N AS LONG
    N = VAL(DecBox.Text)
    HexBox.Text = HEX$(N)
    OctBox.Text = OCT$(N)
END SUB

SUB HexBox_LostFocus ()
    DIM N AS LONG
    N = VAL("&H" + HexBox.Text + "&")
    DecBox.Text = STR$(N)
    OctBox.Text = OCT$(N)
END SUB

SUB OctBox_LostFocus ()
    DIM N AS LONG
    N = VAL("&O" + DecBox.Text + "&")
    HexBox.Text = HEX$(N)
    DecBox.Text = STR$(N)
END SUB

SUB ReturnBtn_Click ()
    UNLOAD BaseForm
END SUB
```

10
Form input

The programs that I have presented so far relied mostly on text boxes to obtain input from the user. This chapter discusses other controls that you can draw on the form to obtain different types of input. Here, you can learn about the following:

- Monitoring text box input.
- Option buttons.
- Check boxes.
- List boxes and Combo boxes.
- Scroll bars.
- The INPUTBOX$ input dialog box.

Short examples are presented for each of the topics listed above, and then a new version of the scientific calculator is shown that uses option buttons, check boxes, and a combo box.

Monitoring text box input

All of the programs presented so far have made no attempt to monitor your keystrokes while you typed in a text box. Visual Basic allows you to regulate what is typed in a text box through the KeyPress event. This event is only generated when a standard ASCII character is typed—this includes control characters, the ENTER (ASCII 13) character, the BACKSPACE (ASCII 8) character, the alphanumeric characters, and the punctuation characters.

The KeyPress procedure-handlers have one parameter, the Integer-typed KeyAscii. The KeyPress empowers you to intercept standard ASCII charac-

ters that you type *before* they appear in the text box (see FIG. 10-1). This enables the KeyPress event-handler to perform one of three tasks:

- Alter the input character.
- Remove the input character.
- Echo the input character without changing it.

The KeyPress event-handler works by examining and manipulating the argument for the KeyAscii parameter, which is passed by reference. This parameter represents the ASCII code for the input character. Typical KeyPress event-handlers perform the following steps:

- Converts the KeyAscii argument to a single-letter string using the Chr$ function. The result can be assigned to either a dynamic string or a single-character string (i.e., a local variable declared with As String * 1).
- Examines the string using a variety of relational and logical operators. Examples are given below, assuming that the local variable C has already been assigned Chr$(KeyAscii):

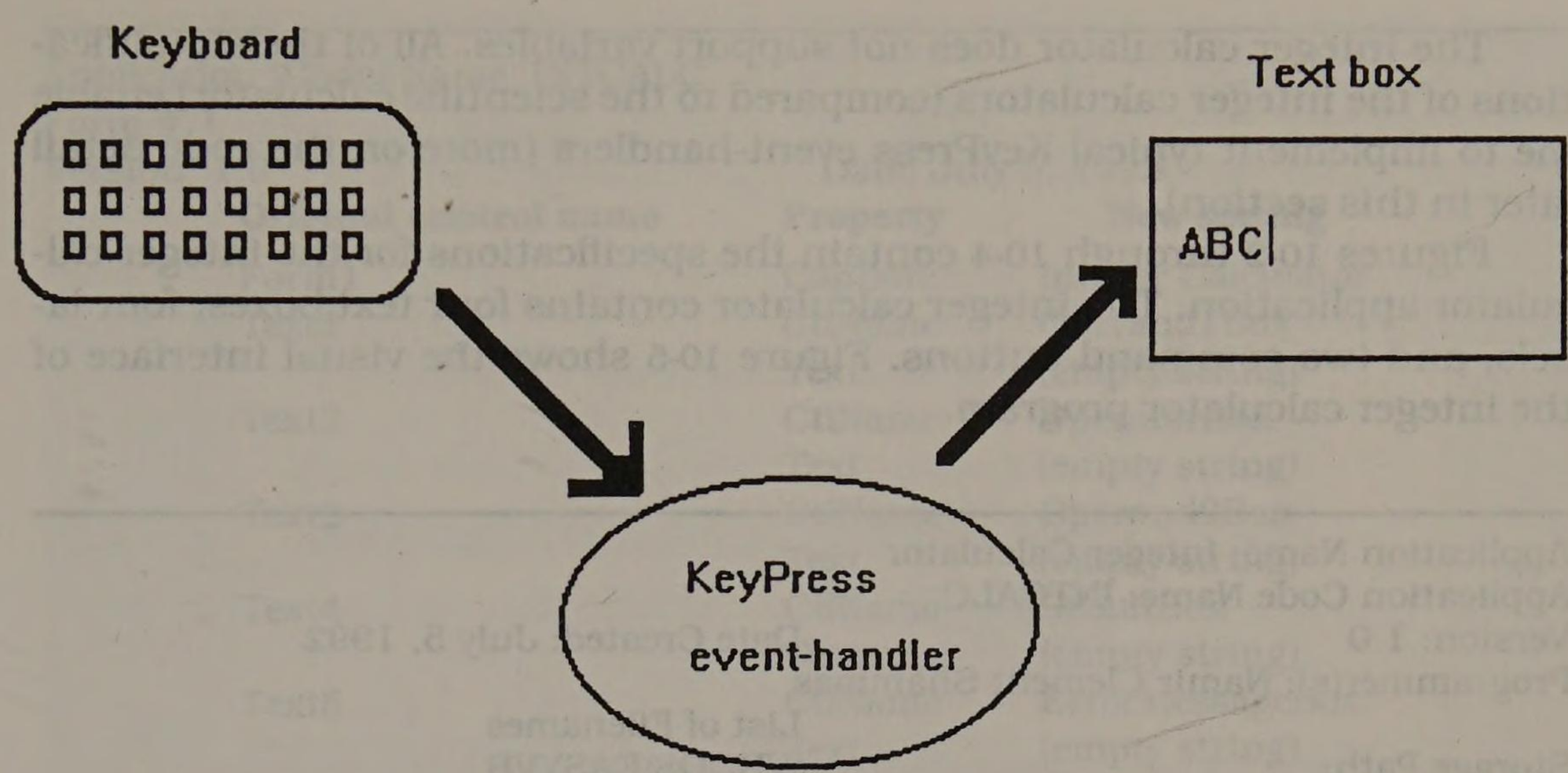
```
IF C >= "0" AND C <= "9" THEN ' is character a digit?
IF (C < "0" AND C > "9") THEN ' is character is a digit?
IF C >= "A" AND C <= "Z" THEN ' is character an uppercase
    letter?
IF C >= "a" AND C <= "z" THEN ' is character a lowercase
    letter?
IF (C >= "a" AND C <= "z") Or (C >= "A" AND C <= "Z")
    THEN ' is a letter?
```

- Takes the appropriate action. If the examined character is satisfactory, the KeyAscii argument is not assigned any new value. As a result, the argument for KeyAscii maintains the original value.

If the examined character must be changed to conform to a certain criteria, then the string variable is first altered and then its ASCII code is reassigned to the argument KeyAscii. Examples are shown below:

```
' make uppercase
C = UCASE$(C)
KeyAscii = ASC(C)

' Make lowercase
C = LCASE$(C)
KeyAscii = ASC(C)
```

☞ If the input character is rejected, the event-handling procedure assigns a 0 to the argument of KeyAscii. As a result, the rejected character that you just typed does not appear in the text box. An example of rejecting an input character is a text box that accepts positive integers (in the decimal base)—any nondigit character is rejected.



10-1 The keystroke interception of KeyPress event-handlers.

Alternate techniques for the above steps might compare the value of KeyAscii with the ASCII code of a character. Examples are shown next:

```
' is character a digit?
IF KeyAscii >= ASC("0") AND KeyAscii <= ASC("9") THEN

' is character not a digit?
IF KeyAscii < ASC("0") OR KeyAscii < ASC("9") THEN
```

☞ **Program note** The KeyPress event is available for other controls. You can alter the input character only for the text box and combo box controls. Other controls can use the KeyPress event to examine the input character and then decide how to respond to that character.

Let's look at an example of an application that monitors the input of text boxes. Next, the integer calculator program is introduced, which is a scaled-down version of the scientific calculator program that I presented earlier in this book. This modified program version has the following features:

- Handles positive integers only.
- Supports the four basic math operations and the modulo operator. The list of valid operator symbols and names are:
  - ~ Addition: +, Add, and Addition.
  - ~ Subtraction: −, Sub, and Subtract.
  - ~ Multiplication: *, Mul, and Multiply.
  - ~ Divide: /, \, Div, and Divide.
  - ~ Modulo: Mod and Modulo
- The Execute button executes the specified operator and places the result in the Result text box. Error messages appear in the Error Message text box.

The integer calculator does not support variables. All of the simplifications of the integer calculators (compared to the scientific calculator) enable me to implement typical KeyPress event-handlers (more on the code detail later in this section).

Figures 10-2 through 10-4 contain the specifications for the integer calculator application. The integer calculator contains four text boxes, four labels, and two command buttons. Figure 10-5 shows the visual interface of the integer calculator program.

Application Name: Integer Calculator
Application Code Name: INTCALC
Version: 1.0 — Date Created: July 5, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	INTCALC.MAK
Form 1	INTCALC.FRM

10-2 The basic specifications for the integer calculator program.

Form # 1 — Form Filename: INTCALC.FRM
Version: 1.0 — Date: July 5, 1992

Control object type	Default CtlName	Purpose
Text Box	Text1	Key in operand 1.
	Text2	Key in operator.
	Text3	Key in operand 2.
	Text4	Display the result.
	Text5	Display error message.
Label	Label1	Label of Text1.
	Label2	Label of Text2.
	Label3	Label of Text3.
	Label4	Label of Text4.
	Label5	Label of Text5.
Command button	Command1	Execute math operation.
	Command2	Quit.

10-3 The basic specifications for the INTCALC.FRM form.

Listing 10-1 contains the code attached to the various controls of the INTCALC.FRM form. The following event-handling procedures are of particular interest:

The Operand1Box_KeyPress and Operand2Box_KeyPress These procedures reject any character that is neither a digit nor the backspace character. The first If statement checks to see whether the argument for KeyAscii is 8 (the

Application (Code) Name: INTCALC
Form # 1
Version: 1.0 Date: July 5, 1992

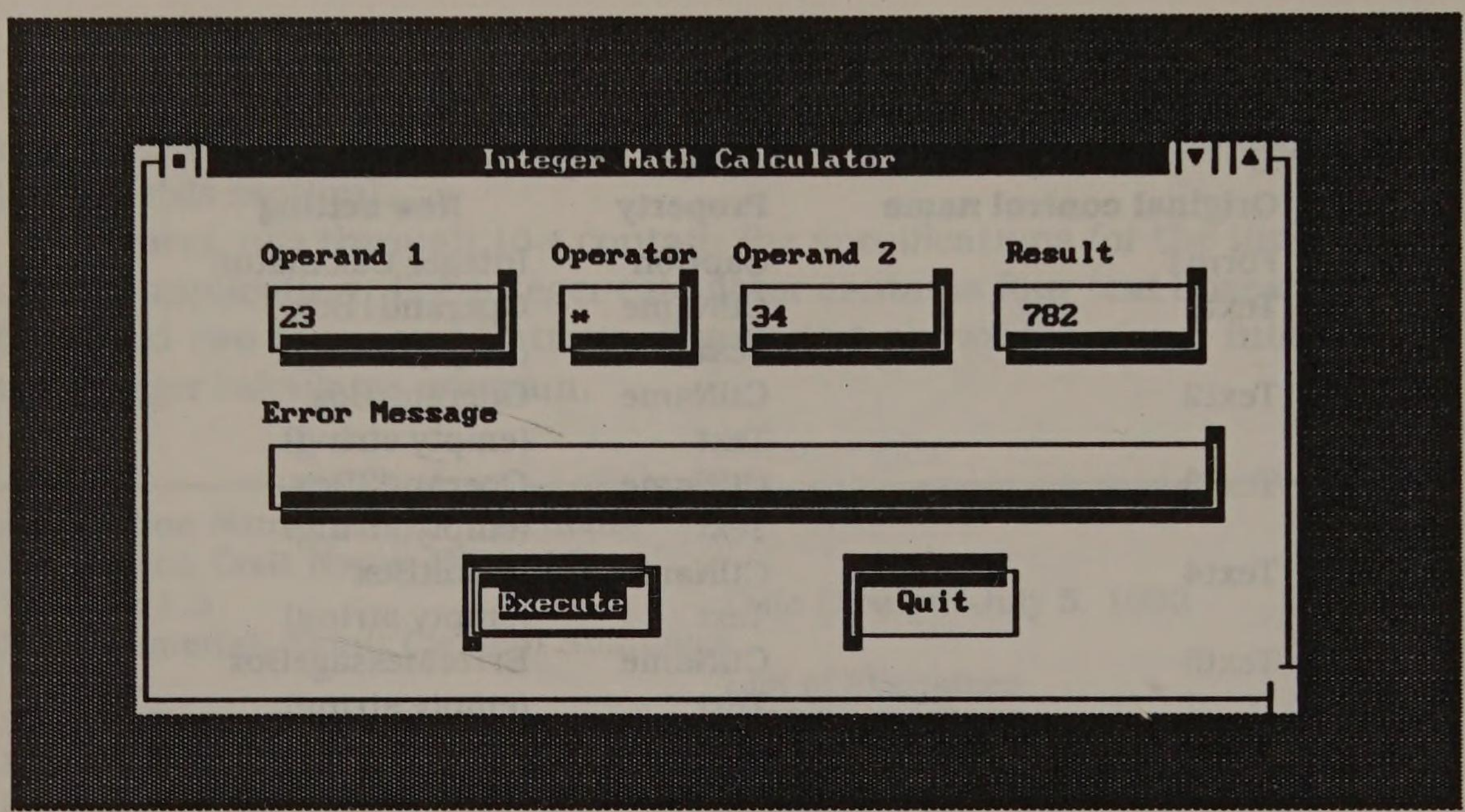
Original control name	Property	New setting
Form1	Caption	Integer Calculator
Text1	CtlName	Operand1Box
	Text	(empty string)
Text2	CtlName	OperatorBox
	Text	(empty string)
Text3	CtlName	Operand2Box
	Text	(empty string)
Text4	CtlName	ResultBox
	Text	(empty string)
Text5	CtlName	ErrorMessageBox
	Text	(empty string)
Label1	CtlName	Operand1Lbl
	Caption	Operand 1
Label2	CtlName	OperatorLbl
	Caption	Operator
Label3	CtlName	Operator2Lbl
	Caption	Operand 2
Label4	CtlName	ResultLbl
	Caption	Result
Label5	CtlName	ErrorMessageLbl
	Caption	Error Message
Command1	CtlName	ExecuteBtn
	Caption	Execute
Command2	CtlName	QuitBtn
	Caption	Quit

10-4 Custom settings for form INTCALC.FRM.

ASCII code for the Backspace character). If the test is positive, the procedure is exited, leaving the argument for KeyAscii unchanged. Removing the first If statement in these procedures causes the backspace character to be rejected as an invalid character. The second If statement examines whether the value of KeyAscii is outside the range for the digit ASCII codes. If the test is positive, the procedure invokes the Beep statement and then assigns a 0 to the KeyAscii argument. This assignment causes the character that you keyed to be rejected.

The OperatorBox_KeyPress This procedure converts the letters that you type into uppercase. The statements in the procedure represent the typical code, described earlier, to alter the user-input.

Run the integer application by pressing the F5 function key. Type the operands in their respective text boxes. While typing an operand,



10-5 A sample session with the integer calculator program.

Listing 10-1 The code attached to the INTCALC.FRM.

```
'$INCLUDE: 'CONSTANT.BI'

SUB ErrorMessageLbl_Click ()
      ErrorMessageBox.Text = ""
END SUB

SUB ExecuteBtn_Click ()
    DIM X AS LONG: DIM Y  AS LONG: DIM Z  AS LONG
    ErrorMessageBox.Text = "" ' clear error message box
    ResultBox.Text = ""       ' clear result box
    X = VAL(Operand1Box.Text)
    Y = VAL(Operand2Box.Text)
    ' examine the content of the OperatorBox text box
    SELECT CASE UCASE$(OperatorBox.Text)
        CASE "+", "ADD"
            Z = X + Y
        CASE "-", "SUB", "SUBTRACT"
            Z = X - Y
        CASE "*", "MUL", "MULTIPLY"
            Z = X * Y
        CASE "/", "\", "DIV", "DIVIDE"
            IF Y <> 0 THEN
                Z = X \ Y
            ELSE
                ErrorMessageBox.Text = "Division by zero error"
                BEEP
```

```
        END IF
    CASE "MOD", "MODULO"
        IF Y <> 0 THEN
            Z = X MOD Y
        ELSE
            ErrorMessageBox.Text = "Modulo of zero error"
            BEEP
        END IF
    CASE ELSE
        ErrorMessageBox.Text = "Invalid operator"
        BEEP
    END SELECT
    ' if the error message box is still empty, there is
    ' no error.  Therefore, convert the variable Z into
    ' a string and display it in ResultBox
    IF ErrorMessageBox.Text = "" THEN
        ResultBox.Text = STR$(Z)
    END IF
END SUB

SUB Label5_Click ()
    ' clear the error message text box
    ErrorMessageBox.Text = ""
END SUB

SUB Operand1Box_GotFocus ()
    DigitCount = 0
END SUB

SUB Operand1Box_KeyPress (KeyAscii AS INTEGER)
    ' is the KeyAscii the ASCII code for
    ' the Backspace character?
    IF KeyAscii = 8 THEN
        EXIT SUB
    END IF
    IF (KeyAscii < ASC("0") OR KeyAscii > ASC("9")) THEN
        BEEP
        KeyAscii = 0
    END IF
END SUB

SUB Operand2Box_KeyPress (KeyAscii AS INTEGER)
    ' is the KeyAscii the ASCII code for
    ' the Backspace character?
    IF KeyAscii = 8 THEN
        EXIT SUB
    END IF
    IF (KeyAscii < ASC("0") OR KeyAscii > ASC("9")) THEN
        BEEP
        KeyAscii = 0
```

Listing 10-1 Continued.

```
    END IF
END SUB

SUB OperatorBox_KeyPress (KeyAscii AS INTEGER)
    DIM C AS STRING * 1
    C = CHR$(KeyAscii) ' convert ASCII code to character
    C = UCASE$(C)      ' convert to uppercase
    KeyAscii = ASC(C)' convert character to ASCII code
END SUB

SUB QuitBtn_Click ()
      END
END SUB
```

press a nondigit key, such as the space bar or a letter. What happens? The corresponding KeyPress procedure beeps at you and rejects your input. Or type the name of an operation, such as add. Notice that the uppercase letters ADD appear in the next box even though you are typing them in lowercase.

Option buttons

Option buttons, also known as radio buttons, are special controls that enable you to provide input by selecting a logical state. Option buttons usually work in groups of two or more. You can only select one option button in a group—the other option buttons in the same group are automatically deselected. The option buttons represent logically Ored states; each state is a mutually exclusive alternative. Option buttons resemble the button of a radio, which enables you to select only one station at a time (whence the nickname radio button). Option buttons should be grouped in a form. This grouping is carried out in one of two ways: using control arrays or drawing picture boxes.

A control array drawn in the form. Each group of option buttons shares the same name but has a distinct setting for the Index property. The Visual Basic interface allows you to draw an option button control array in two ways: explicit and implicit. In the explicit method, you draw the first member of the option button array and assign a value to the Index property. This tells the Visual Basic interface system that you started a new control array. When you draw the rest of the control arrays, type in the same control name; the Index setting is automatically assigned by the Visual Basic interface.

In the second method, you draw the first control as a nonarray object—

so far the Visual Basic interface thinks you are drawing a single control. When you draw the second option button, type in the same control name. This duplicate name causes the Visual Basic interface to ask you whether or not you want to create a control array. Select the Yes button from the inquiring dialog box. The rest of the buttons are created with the same name; the Index setting is automatically assigned by the Visual Basic interface. In either method, enter an Index setting if you wish to bypass the incremented control array numbers that are provided by the Visual Basic interface.

☞ **Program note** You can use different control names for option buttons in a frame or a picture box and still have the single-alternative feature enforced by Visual Basic. Visual Basic treats such option buttons as a group, since they are located in a grouping control.

Draw a frame or a picture box and then draw the option buttons inside the frame. The option buttons are drawn as control arrays, because the Visual Basic interface understands that these option buttons are related.

Option buttons possess a number of properties that define their visual appearance and behavior. Some properties, such as Caption, CtlName, Enabled, and Visible, resemble those of command buttons. The most relevant option button property is Value. This is a logical property that determines whether or not an option button is selected. The general syntax for the Value property is shown below:

```
[form.] [optionButton.]Value [=Boolean%]
```

When the Value property of an option button is TRUE (−1), the button is selected and the Value of all the other option buttons in the same group are set to FALSE (0). The Value property can be set and queried both at design and run time.

Option buttons can be selected by either clicking the mouse on them or by tabbing to the sought option and then pressing the SPACEBAR.

Here is an example of a Visual Basic program that uses option buttons. The program is an enhanced version of the APP1.MAK project that displays the current date and time. The new version includes two groups of option buttons. Each group is located inside a frame. These option button groups enable you to select the date and time display formats. The date format options are DD-MM-YYYY, MM-DD-YYYY, and YYYY-MM-DD. The time formats are AM/PM and 24 hours.

Figures 10-6 through 10-8 contain the specifications for the OPTION.FRM form. The date-format and time-format option buttons have the array control name of DateFormatOpt and TimeFormatOpt, respectively. Figure 10-9 shows a sample session with the OPTION.MAK project.

Listing 10-2 contains the code attached to the OPTION.FRM form. There are three event-handling procedures:

Application Name: Option Button Demonstration	
Application Code Name: OPTION	
Version: 1.0	Date Created: July 5, 1992
Programmer(s): Namir Clement Shammas	
	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	OPTION.MAK
Form 1	OPTION.FRM

10-6 The general specifications of the OPTION.MAK project file.

Form # 1	Form Filename: OPTION.FRM
Version: 1.0	Date: July 5, 1992

Control object type	Default CtlName	Purpose
Command Button	Command1	Displays the date and time.
	Command2	Exits the test program.
Text Box	Text1	Displays the date and time.
Frame	Frame1	Contains the date format option buttons.
	Frame2	Contains the time format option buttons.
Option Button	Option1	0 - DD-MM-YYYY date format. 1 - MM-DD-YYYY date format. 2 - YYYY-MM-DD date format.
	Option2	0 - AM/PM time format. 1 - 24 hour time format.

10-7 The list of controls in the OPTION.FRM form file.

- The Form_load procedure initializes the form by selecting the second date-format option button and the second time-format option button.
- The NowBtn_Click procedure displays the current date and time. The procedure uses the local string-typed variable S to build the date and time image. Two extended IF-THEN-ELSEIF statements are used to examine the Value property of the DateFormatOpt and TimeFormatOpt option button arrays. The predefined NOW and FORMAT$ functions produce a string that contains the properly formatted date

Application (Code) Name: OPTION

Form # 1

Version: 1.0 Date: July 5, 1992

Original control name	Property	New setting
Form	Caption	Option Button Demo
Command1	CtlName	NowBtn
	Caption	Now
Command2	CtlName	QuitBtn
	Caption	Quit
Text1	CtlName	TextBox
	Caption	(empty string)
Frame1	CtlName	DateFormatFrm
	Caption	Date Format
Frame2	CtlName	TimeFormatFrm
	Caption	Time Format
Option1	CtlName	DateFormatOpt
	Index	0
	Caption	DD-MM-YYYY
	CtlName	DateFormatOpt
	Index	1
	Caption	MM-DD-YYYY
	CtlName	DateFormatOpt
	Index	2
	Caption	YYYY-MM-DD
Option2	CtlName	TimeFormatOpt
	Index	0
	Caption	AM/PM
	CtlName	TimeFormatOpt
	Index	1
	Caption	24 Hr

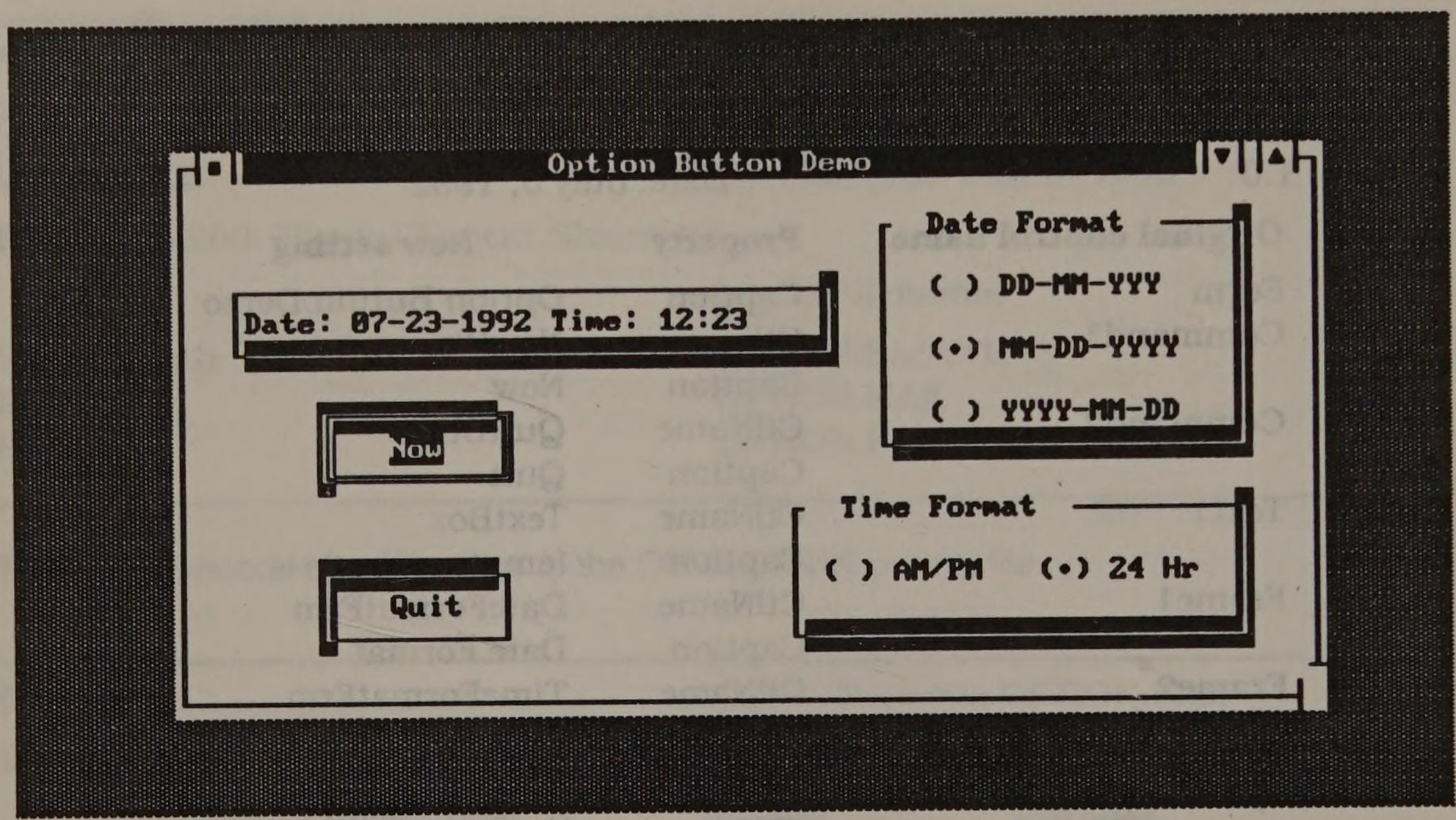
10-8 The custom settings for the controls in the OPTION.FRM form.

or time. Both IF statements have ELSE clauses that read the date and time from the DATE$ and TIME$ functions, respectively, when all of the option buttons are turned off in either group. The last procedure statement assigns the local string S to the Text property of TextBox.

- The QuitBtn procedure exits the test program.

Notice that the attached code is void of any user-defined procedure that handles selecting and deselecting an option button; all that work is handled by the run-time system.

Run the program and select the various date and time format option buttons. After each combination of time and date option buttons, press the



10-9 A sample session with the OPTION.MAK project.

Listing 10-2 The code attached to the OPTION.FRM form.

```
'$INCLUDE: 'CONSTANT.BI'

SUB Form_Load ()
    DateFormatOpt(1).Value = TRUE
    TimeFormatOpt(1).Value = TRUE
END SUB

SUB NowBtn_Click ()
    DIM S AS STRING
    IF DateFormatOpt(0).Value THEN
        S = FORMAT$(NOW, "dd-mm-yyyy")
    ELSEIF DateFormatOpt(1).Value THEN
        S = FORMAT$(NOW, "mm-dd-yyyy")
    ELSEIF DateFormatOpt(2).Value THEN
        S = FORMAT$(NOW, "yyyy-mm-dd")
    ELSE
        S = DATE$
    END IF
    S = "Date: " + S + " Time: "
    IF TimeFormatOpt(0).Value THEN
        S = S + FORMAT$(NOW, "hh:mm AM/PM")
    ELSEIF TimeFormatOpt(1).Value THEN
        S = S + FORMAT$(NOW, "hh:mm")
    ELSE
        S = S + TIME$
```

```
    END IF
    TextBox.Text = S ' assign new text
END SUB

SUB QuitBtn_Click ()
    END
END SUB
```

Now command button to view the date and time as specified. When you finish examining the option buttons, click on the Quit button to exit.

Check boxes

Check boxes are controls that can be switched on or off. Check boxes can be used individually or in a group to manipulate a series of conditions. The on/off state of each check box is independent of that of other check boxes. You can substitute a group of two option buttons with a single check box, when the alternate choices are clear; otherwise, it might be better to use the option buttons.

The check box control possesses a number of properties, such as Caption, CtlName, Enabled, and Visible, that resemble those of other controls. Like the option button, the check box has the Value property. However, unlike the option button, the Value property of a check box takes one of three values. The general syntax of using the Value property of a check box is:

[form.] [checkBox.]Value [=enumeratedSetting%]

The settings of the Value property are 0, 1, and 2. When the Value property is set to 0, the check box is cleared. Assigning 1 to the Value property selects the check box, placing an X inside the box. When the Value property is assigned 2, the box is grayed. The value of 2 can be used to implement a semidisabled state, because the check box is neither checked nor unchecked.

The following short example demonstrates how check boxes are used to alter the display of the date and time. This demonstration program resembles the last one that I presented. The main difference is the use of two check boxes instead of two option button groups. The first check box enables you to toggle between the AM/PM and the 24-hour time formats. The second check box specifies whether or not the current date is included in the TextBox when you click on the Now command button.

Figures 10-10 through 10-12 list the specifications for this demonstration program. The check boxes have the default values of 0—they are clear when the program starts running. Figure 10-13 shows a sample session with the test program.

Listing 10-3 shows the two procedures attached to the controls in CHECKBOX.FRM. The NowBtn_Click procedure builds the date and time

Application Name: Check Box Demonstration

Application Code Name: CHECKBOX

Version: 1.0 Date Created: July 5, 1992

Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	CHECKBOX.MAK
Form 1	CHECKBOX.FRM

10-10 The general specifications for the CHECKBOX.MAK project.

Form # 1 Form Filename: CHECKBOX.FRM

Version: 1.0 Date: July 5, 1992

Control object type	Default CtlName	Purpose
Command Button	Command1	Displays the date and time.
	Command2	Exits the test program.
Text Box	Text1	Displays the date and time.
Check Box	Check1	Displays 24 hour time format.
	Check2	Displays the date.

10-11 The general specifications for the CHECKBOX.FRM form.

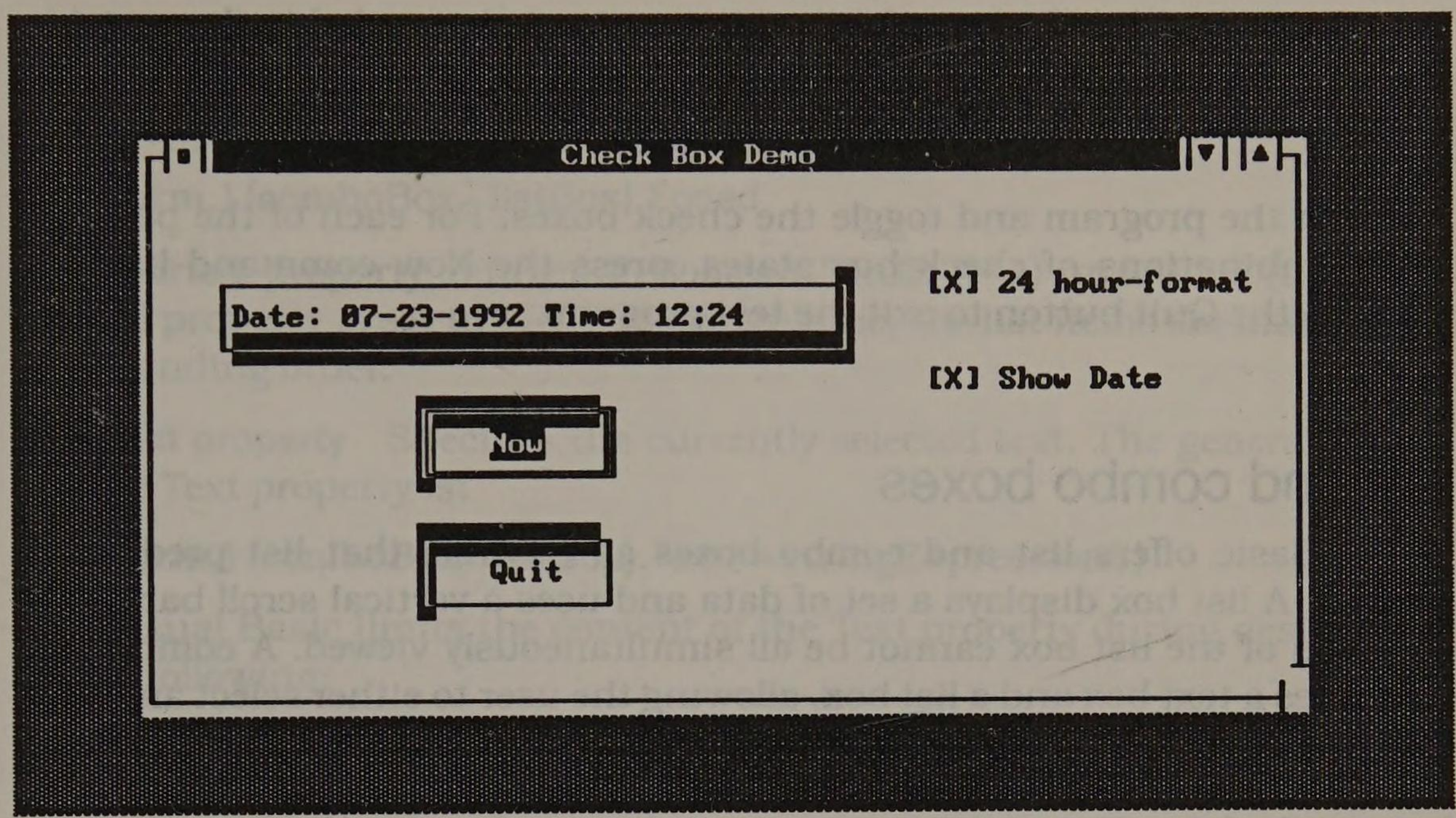
Application (Code) Name: CHECKBOX

Form # 1

Version: 1.0 Date: July 5, 1992

Original control name	Property	New setting
Form	Caption	Check Box Demo
Command1	CtlName	NowBtn
	Caption	Now
Command2	CtlName	QuitBtn
	Caption	Quit
Text1	CtlName	TextBox
	Caption	(empty string)
Check1	CtlName	TimeFormatChk
	Caption	24 Hour
Check2	CtlName	ShowDateChk
	Caption	Show Date

10-12 The new settings for the CHECKBOX.FRM form.



10-13 A sample session with the CHECKBOX program.

Listing 10-3 . The code attached to the objects in CHECKBOX.FRM.

```
Sub NowBtn_Click ()
  Dim S As String
  S = ""
  If ShowDateChk.Value = 1 Then
    S = "Date: " + Date$ + " "
  End If
  S = S + "Time: "
  If TimeFormatChk.Value = 1 Then
    S = S + Format$(Now, "hh:mm")
  Else
    S = S + Format$(Now, "hh:mm AM/PM")
  End If
  TextBox.Text = S ' assign new text
End Sub

Sub QuitBtn_Click ()
  End
End Sub
```

string using a local string variable S. The procedure uses an If-Then statement to compare the Value property of the Show Date check box with 1. If the Show Date check box is selected, the test is positive and the date information is assigned to the string. The program also uses an IF-THEN-ELSE statement to examine the Value property of the time format check box. If

that property is 1, a 24-hour format time string is appended to the variable S; otherwise, an AM/PM format time string is appended. The last statement in the procedure assigns the contents of variable S to the Text property of TextBox.

Run the program and toggle the check boxes. For each of the possible four combinations of check box states, press the Now command button. Click on the Quit button to exit the test program.

List and combo boxes

Visual Basic offers list and combo boxes as controls that list predefined choices. A list box displays a set of data and uses a vertical scroll bar if the contents of the list box cannot be all simultaneously viewed. A combo box combines a text box and a list box, allowing the user to either select an entry from the list or type in an input.

List and combo boxes have a number of properties, events, and methods in common. Before discussing these common traits, let me start with an important property that is unique to the combo boxes, but not to the lists boxes. This is the Style property that determines the type and behavior of a combo box at run time. This property can be set only at design time. During program execution the Visual Basic code can only access this property. The general syntax for the Style property is:

```
[form.]comboBox.Style
```

The Style property has one of three values, namely, 0, 1, and 2. The default setting for the Style property is 0. This setting makes the combo box behave as a Dropdown combo box. When you click on the down arrow the associated list appears, and you can then either select a list entry or type in your input in the edit area (the text box portion of the combo box). The list is closed after you make a selection.

When you set the Style property to 1, you have what is called a Simple combo box. This type of box always displays the list and the edit area. The default size of the Simple combo box is such that it hides the list. Alter the Height property to show more of the list.

Setting the Style property to 2 gives you a Dropdown list which includes the list but no edit area. This type of combo box forces you to select an item in the list.

When the Style property is set to 0 or 2, you can open the list by either clicking on the down-arrow button or by pressing the ALT+DOWN keys when the combo box has the focus.

The list and combo boxes share a number of properties that enable you to maintain and select items from their lists. These items are stored as an array of strings with the lower index of 0. The properties that manage list selections are:

The Sorted property Determines whether or not the list items are alphabetically sorted. The Sorted property can be set only during design time. The general syntax for this run-time read-only property is:

[form.] {comboBox ¦ listBox}.Sorted

The Sorted property returns a Boolean TRUE (−1) or FALSE (0). If the Sorted property is set to TRUE at design time, the list items are maintained in ascending order.

The Text property Specifies the currently selected text. The general syntax for the Text property is:

[form.] {comboBox ¦ listBox}.Text [=stringExpression$]

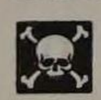
Visual Basic limits the content of the Text property during design time to the following:

- A list box has an empty string in the Text property.
- A combo box can have either an empty string or the default control name supplied by Visual Basic.
- You can assign values to the Text property of the combo box when the Style setting is either 0 or 1. Assigning a string to the Text property when the Style setting is 2 results in a run-time error, because you can only obtain a selection from the list.

The List property Represents an array of strings that contains the list data. The general syntax for the List property is:

[form.] {comboBox ¦ listBox}.List(index%) [=stringExpression$]

Using the List property you can overwrite a current list item.



Warning Overwriting list items when the Sorted property is TRUE might corrupt the order in the list. Consequently, you need to have sorting procedures that reorder the items in the list before you add new data items.

The ListIndex property Indicates the index of the currently selected item. The value of the ListIndex for a list box ranges from −1 to the number of data items minus one (that is, ListCount −1). The range of the ListIndex for Combo boxes is −1 to the number of data items minus one. The −1 setting accounts for selections that do not have a matching list member. The general syntax for this run-time read-only property is:

[form.] {comboBox ¦ listBox}.ListIndex [=index%]

The ListCount property Maintains the current number of items in the list. The general syntax for this run-time read-only property is:

[form.] {comboBox ¦ listBox}.ListCount

The list box recognizes the Click and DblClick events. Clicking on an item

in a list box selects that item. Microsoft recommends the following guidelines on using the single and double click events:

- Each list should be accompanied by a command button. Thus, a single click in the list box selects an item, and a click on the associated command button performs a task using the selected list item.
- Double clicking on a list item should have the same effect as single clicking a list item followed by clicking on the associated command button. Thus, double clicking is a shortcut action.

The combo box recognizes the Click, DblClick, Change, and DropDown events (among others). The Click event is generated when you select a list item. The Change event is recognized (when the Style property is set to 0 or 1) when you edit the contents of the text box. The DropDown event is recognized when the Style property is 0 or 2.

The list and combo boxes have the following methods that are used to add and remove items from the lists:

The ADDITEM method Inserts new data items in the list. The general syntax for the AddItem method is:

```
[form.] {controlBox ¦ listBox}.ADDITEM item$ [,index%]
```

The index% is optional. If it is omitted and the Sorted property is FALSE, the new item is appended to the list. If the Sorted property is TRUE, the new item is inserted in its proper place.

When the Sorted property is TRUE, you must omit the index% value, lest you corrupt the order of the items in the list.

When the Sorted property is FALSE, the value of index% can be in the range of 0 to ListCount. If the value of index% is less than ListCount, the list items are shifted upwards from the index% location.

The REMOVEITEM method Removes an item from the list. The general syntax for the REMOVEITEM method is:

```
[form.] {controlBox ¦ listBox}.REMOVEITEM index%
```

The values for the index% range from 0 to ListCount – 1.

Let's look at a simple application that uses the combo box. I have modified the Option Box Demonstration program such that the date format is selected by country. The time format option buttons have been dropped out of the combo box demonstration program.

Figures 10-14 through 10-16 show the specifications for the combo box demonstration program. Notice the new settings of the combo box in FIG. 10-16. The Sorted property is set to TRUE and the Style is set to 2 (Dropdown list box). Figure 10-17 shows a sample session with the combo box demonstration program.

Application Name: Combo Box Demonstration

Application Code Name: COMBOBOX

Version: 1.0 Date Created: July 5, 1992

Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	COMBOBOX.MAK
Form 1	COMBOBOX.FRM

10-14 The general specifications for the COMBOBOX.MAK project.

Form # 1 Form Filename: COMBOBOX.FRM

Version: 1.0 Date: July 5, 1992

Control object type	Default CtlName	Purpose
Command Button	Command1	Displays the date and time.
	Command2	Exits the test program.
Text Box	Text1	Displays the date and time.
Combo Box	Combo1	Displays date format by country.

10-15 The general specifications for the COMBOBOX.FRM form.

Application (Code) Name: COMBOBOX

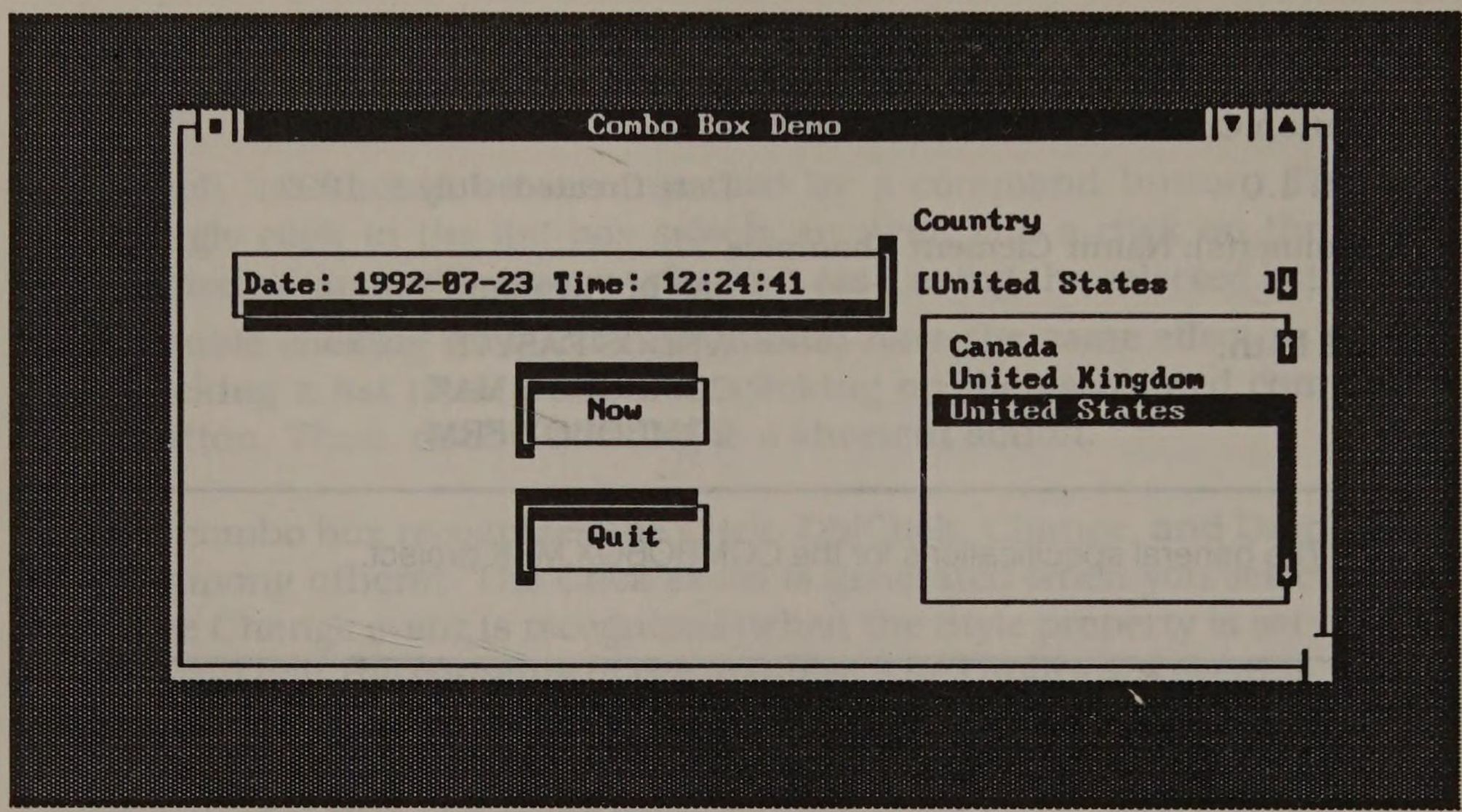
Form # 1

Version: 1.0 Date: July 5, 1992

Original control name	Property	New setting
Form	Caption	Combo Box Demo
Command1	CtlName	NowBtn
	Caption	Now
Command2	CtlName	QuitBtn
	Caption	Quit
Text1	CtlName	TextBox
	Caption	(empty string)
Combo1	CtlName	CountryCmb
	Caption	(empty string)
	Sorted	TRUE
	Style	2

10-16 The new settings for the COMBOBOX.FRM form.

Listing 10-4 contains the code attached to the controls of the COMBOBOX.FRM form. The procedures are:



10-17 A sample session with the COMBOBOX program.

The Form_Load procedures Add the data to the combo list box using the AddItem method. Because the Sorted property is set to TRUE, no insertion index is specified.

The NowBtn_Click procedure Displays the date and time in control TextBox. The date format is selected based on the country you specify. The CountryCmb.Text returns the selected text. This value is used in a Select-Case statement to compare the selected text with the list items. These items are retrieved using the List property. When the selected text matches a list item, the corresponding date format is used to yield a date string. The time is appended to the date/time string by simply calling the Time$ function.

The QuitBtn_Click procedure Exits the demonstration program.

Vertical and horizontal scroll bars

The vertical and horizontal scroll bars are controls that enable you to quickly go through a list of data. Using scroll bars is typical of applications and utilities that use windows. The vertical and horizontal scroll bars are very similar, differing only in their visual orientation. Each scroll bar has two arrow boxes, one at each end. The minimum and maximum ends of a vertical scroll bar are located at the top and bottom ends, respectively. The minimum and maximum ends of a horizontal scroll bar are located at the left and right ends, respectively.

Scroll bars represent integer values. The following properties specify the

Listing 10-4 The code attached to the controls of the COMBOBOX.FRM form.

```
'$INCLUDE: 'CONSTANT.BI'

SUB Form_Load ()
    ' add items into combo box
    CountryCmb.ADDITEM "United States"
    CountryCmb.ADDITEM "United Kingdom"
    CountryCmb.ADDITEM "Canada"
END SUB

SUB NowBtn_Click ()
    DIM S AS STRING
    SELECT CASE CountryCmb.Text
        CASE CountryCmb.List(2) ' UNITED STATES
            S = FORMAT$(NOW, "mm-dd-yyyy")
        CASE CountryCmb.List(1) ' CANADA
            S = FORMAT$(NOW, "yyyy-mm-dd")
        CASE CountryCmb.List(0) ' UNITED KINGDOM
            S = FORMAT$(NOW, "dd-mm-yyyy")
        CASE ELSE
            S = DATE$
    END SELECT
    TextBox.Text = "Date: " + S + " Time: " + TIME$
END SUB

SUB QuitBtn_Click ()
    END
END SUB
```

range of values, the current scroll bar value, and the degree of change in the latter value:

The Min property Determines the lower range of values for a scroll bar. This property can be set during design and run times, and can be accessed during run time. The general syntax for the Min property is:

[form.] {hscrollBar ¦ vscrollBar}.Min [=limit%]

The values for the Min property range between −32768 and 32767. The default value is 0.

The Max property Determines the upper range of values for a scroll bar. This property can be set during design and run times, and can be accessed during run time. The general syntax for the Max property is:

[form.] {hscrollBar ¦ vscrollBar}.Max [=limit%]

The values for the Max property range between −32768 and 32767. The default value is 32767.

The Value property Sets or determines the current scroll bar value or reading. This property can be set during design and run times, and can be accessed during run time. The general syntax for the scroll bar Value property is:

[form.] {hscrollBar ¦ vscrollBar}.Value [=setting%]

The values for the Value property range are defined by the Min and Max properties and determines the location of the scroll box.

The SmallChange property Sets and determines the change in the Value property when you click the arrows at either end of the scroll bar. The general syntax for the SmallChange property is:

[form.] {hscrollBar ¦ vscrollBar}.SmallChange [=change%]

The values for the SmallChange property range from 1 to 32767, with a default setting of 1.

The LargeChange property Sets and determines the change in the Value property when you click above or below the scroll box. The general syntax for the LargeChange property is:

[form.] {hscrollBar ¦ vscrollBar}.LargeChange [=change%]

The values for the LargeChange property range from 1 to 32767, with a default setting of 1.

The Change event is very important in detecting moving the scroll box. By retrieving the Value property, you are able to update the information in a related control.

Let's see a simple example for a scroll bar. The next Visual Basic program is a modification of the APP2.MAK project that offered a count-down timer. This new version contains a count-down display text box, a vertical scroll bar, a count-down command button, and a quit button. The vertical scroll bar allows you to specify the starting countdown time. The scroll bar values range from 0 to 100. When you click on the count-down button, the text box displays the count-down time. At the same time, the scroll box is updated to reflect the current time—the scroll box slowly creeps upward until it reaches the upper end of the scroll bar. While the program is counting down, the command buttons are disabled. When the count-down is finished, the command buttons are enabled.

Figures 10-18 through 10-20 contain the specifications for the SCROLLBA.MAK project file and the SCROLLBA.FRM form file. Notice that there is only one new setting for the scroll bar—a new control name. The new settings for the Min, Max, Value, SmallChange, and LargeChange properties are made in the Form_Load procedure. Also notice the MinLbl and MaxLbl captions; they are "N Sec" and "M sec" respectively. These are dummy captions. The Form_Load procedure sets new captions that reflect the actual settings for the Min and Max properties. Figure 10-21 shows a sample session with the scroll bar demonstration program.

Application Name: Scroll Bar Demonstration
Application Code Name: SCROLLBA
Version: 1.0 Date Created: July 5, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	SCROLLBA.MAK
Form 1	SCROLLBA.FRM

10-18 The general specifications for the SCROLLBA.MAK project.

Form # 1 Form Filename: SCROLLBA.FRM
Version: 1.0 Date: July 5, 1992

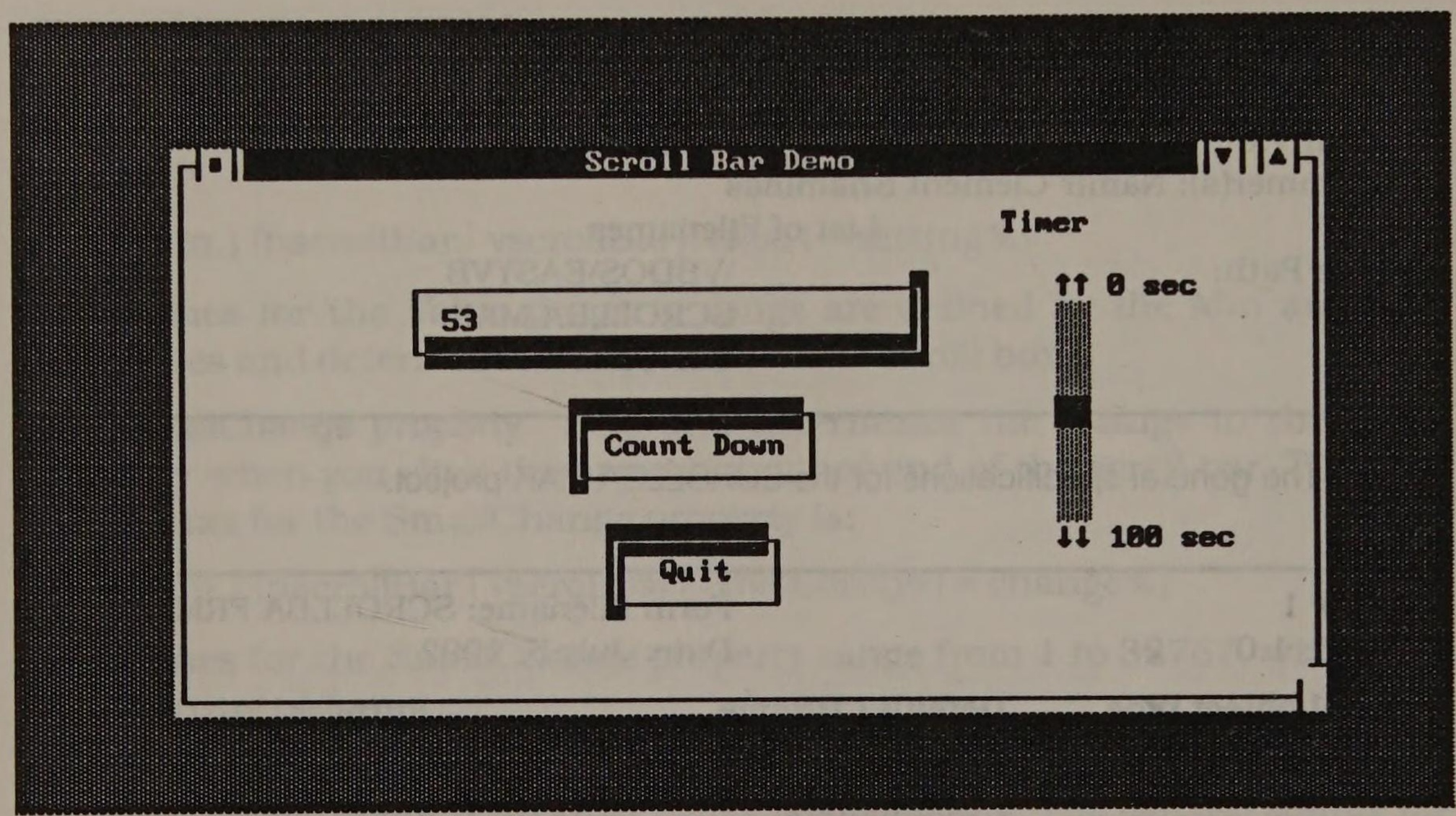
Control object type	Default CtlName	Purpose
Command Button	Command1	Activates the count-down timer.
	Command2	Exits the test program.
Text Box	Text1	Displays the date and time.
Vertical Scroll Bar	VScroll1	Sets the timer.
Label	Label1	Scroll bar label.
	Label2	Minimum scroll bar value label.
	Label3	Maximum scroll bar value label.

10-19 The general specifications for the SCROLLBA.FRM form.

Application (Code) Name: SCROLLBA
Form # 1
Version: 1.0 Date: July 5, 1992

Original control name	Property	New setting
Form	Caption	Scroll Bar Demo
Command1	CtlName	TimerBtn
	Caption	Count Down
Command2	CtlName	QuitBtn
	Caption	Quit
Text1	CtlName	TextBox
	Caption	(empty string)
VScroll1	CtlName	TimeScl
Label1	CtlName	TimerLbl
	Caption	Timer Setting
Label2	CtlName	MinLbl
	Caption	N sec
Label3	CtlName	MaxLbl
	Caption	M sec

10-20 The new settings for the SCROLLBA.FRM form.



10-21 A sample session with the SCROLLBA program.

Listing 10-5 shows the code attached to the SCROLLBA.FRM and its controls. The general declarations section declares two constants, namely, MIN_VALUE and MAX_VALUE, that define the range of scroll bar values. Also notice the following event-handling procedures:

The TimeScl_Change procedure Updates the content of the text box when the scroll box moves.

The Form_load procedure Performs a good amount of initialization. This includes assigning new settings to the Min, Max, Value, SmallChange, and LargeChange properties. The SmallChange property is reassigned the default setting—I place the assignment to include all of the scroll bar properties mentioned earlier. The new settings for most of the above scroll bar properties are based on the MIN_VALUE and MAX_VALUE constants. The Form_Load procedure also initializes the content of the text box. Finally, the procedure assigns new captions for the MinLbl and MaxLbl labels. These captions reflect the values of the MIN_VALUE and MAX_VALUE constants.

☞ **The TimerBtn_Click procedure** Handles the countdown process. The total number of seconds to countdown is taken from the Value property of the scroll bar. Notice that the procedure does not explicitly update the contents of the text box, even though the text box shows the decreasing number of seconds during the countdown. What makes the text box update the time automatically? The answer is procedure TimeScl_Change. Whenever the TimeScl.Value property changes, the contents of the text box are accordingly updated.

Run the scroll bar demo program and experiment with the scroll bar.

Listing 10-5 The code attached to the SCROLLBA.FRM form and its controls.

```
'$INCLUDE: 'CONSTANT.BI'
CONST MIN_VALUE = 0, MAX_VALUE = 100

SUB Form_Load ()
    ' set the Min, Max, Value, SmallChange, and the
    ' LargeChange properties of the scroll bar
    TimeScl.Min = MIN_VALUE
    TimeScl.Max = MAX_VALUE
    TimeScl.Value = MAX_VALUE \ 10
    TimeScl.SmallChange = 1
    TimeScl.LargeChange = MAX_VALUE \ 10
    ' set the Text property of TextBox
    TextBox.Text = STR$(TimeScl.Value)
    ' Assign captions to the scroll bar labels to
    ' match the program's settings
    MinLbl.Caption = FORMAT$(MIN_VALUE) + " sec"
    MaxLbl.Caption = FORMAT$(MAX_VALUE) + " sec"
END SUB

SUB QuitBtn_Click ()
    END
END SUB

SUB TimerBtn_Click ()
      DIM Count AS DOUBLE: DIM Total  AS DOUBLE: DIM Diff  AS DOUBLE
      TextBox.Text = STR$(TimeScl.Value)
      Total = TimeScl.Value
      IF Total = 0 THEN EXIT SUB
      QuitBtn.Enabled = FALSE
      TimerBtn.Enabled = FALSE
      Count = TIMER
      DO WHILE (TIMER - Count) <= Total
            Diff = INT(Total - (TIMER - Count))
            ' update the scroll box location
            IF Diff >= 0 THEN
              TimeScl.Value = Diff
              ' refresh scroll bar?
              IF (INT(Diff) MOD TimeScl.LargeChange) = 0 THEN
                TimeScl.REFRESH
              END IF
            END IF
      LOOP
      BEEP
      TimerBtn.Enabled = TRUE
      QuitBtn.Enabled = TRUE
END SUB

SUB TimeScl_Change ()
    TextBox.Text = STR$(TimeScl.Value)
END SUB
```

Press the Count Downbutton and watch the countdown process. You can alter the MAX_VALUE constant and rerun the program. Notice the new maximum value scroll bar label. It reflects the new value of constant MAX_VALUE.

The INPUTBOX$ and MSGBOX dialog boxes

Input boxes are common types of modal dialog boxes that prompt you to enter a value as a string. The input box contains an edit text box, an OK button, and a Cancel button. If you click the OK button or press the ENTER key you confirm the content of the edit text box. By contrast, if you click the Cancel button you exit the dialog box and disregard the content of the edit box.

Visual Basic offers the INPUTBOX$ function that implements a modal input dialog box. The INPUTBOX$ function returns the content of the edit box if you click the OK button or press the ENTER key. It returns an empty string if you click the Cancel button. As a modal dialog box, the INPUTBOX$ does not allow you to switch to another window until you first close it. The general syntax for the INPUTBOX$ functions is:

```
INPUTBOX$ (prompt$ [, title$ [, default$ [, xpos%, ypos%]]])
```

The prompt$ is the prompting message string. This string can be up to 255 characters long and can contain CHR$(13)+CHR$(10) substrings to create line breaks.

The optional title$ parameter is a title string. When title$ is omitted, the input box appears with no title.

The default$ parameter supplies the default content of the edit text box.

The xpos% and ypos% parameters specify the location of the upper right-hand corner of the dialog box from the upper right-hand screen corner. If these parameters are omitted, the dialog box appears horizontally centered and vertically positioned at about one third the way down the screen.

Visual Basic implements the MSGBOX function and the MSGBOX statement. The function differs from the statement in that it returns an integer value. However, the MSGBOX function is identical to the MSGBOX statement in every other aspect. As the name suggests, both forms of the MSGBOX convey a message to the user and display one or more buttons. The MSGBOX function returns a value that indicates the button you clicked as a response to the message, which is a type of input. The general syntax for the MSGBOX function and statement is:

```
MSGBOX (msg$ [, type% [, title$]])
```

The msg$ parameter specifies the message text displayed by MSGBOX. The msg$ string can be up to 1024 characters (the extra characters are truncated), assuming that the first 255 characters contain at least one space. If not, only the first 255 characters are displayed and the rest of the msg$ string is truncated. The MSGBOX automatically wraps the text of string msg$. You can make new lines by inserting the CHR$(13)+CHR$(10) substrings in the string msg$.

The type% parameter specifies the number and type of buttons, the accompanying icons, and the default button. Table 10-1 shows the basic values

Table 10-1 The basic values for the type% parameter in the MSGBOX.

Group	Value	Purpose
Buttons	0	Displays OK button.
	1	Displays OK and Cancel buttons.
	2	Displays Abort, Retry, and Ignore buttons.
	3	Displays Yes, No, and Cancel buttons.
	4	Displays Yes and No buttons.
	5	Displays Retry and Cancel buttons.
Default	0	First button is default.
	256	Second button is default.
	512	Third button is default.

Table 10-2 The integer results returned by the MSGBOX function.

Value	Meaning
1	OK pressed
2	Cancel pressed
3	Abort pressed
4	Retry pressed
5	Ignore pressed
6	Yes pressed
7	No pressed

for the type% parameter, grouped by purpose. The argument for the type% parameter can combine one value from each group. For example, to have a MSGBOX display the OK and Cancel buttons (value of 1) and to make the second button the default (value of 256), you need to supply the value of 257 (equal to 1 + 256) to the type% argument.

The title$ string enables you to specify a title for the MSGBOX. When omitted, no title appears.

Table 10-2 shows the integer results returned by the MSGBOX function. These values are used as a form of input using the MSGBOX function.

Next, is a short example for the INPUTBOX$ and MSGBOX statements. The next program is a modified version of the last one—the countdown timer. The new version uses the INPUTBOX$ to obtain the countdown time when you press the Count Down button. If you enter zero or a negative value, a message box appears to tell you that your input value is replaced with 15 (seconds).

Figures 10-22 through 10-24 show the specifications for the project file INPUTBOX.MAK and the form file INPUTBOX.FRM. Figure 10-25 shows a sample session with the test program.

Listing 10-6 shows the code attached to the INPUTBOX.FRM. The Timer_Click procedure contains the INPUTBOX$ and MSGBOX state-

Application Name: INPUTBOX$ and MSGBOX Demonstration
Application Code Name: INPUTBOX
Version: 1.0 Date Created: July 5, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	INPUTBOX.MAK
Form 1	INPUTBOX.FRM

10-22 The general specifications for the INPUTBOX.MAK project.

Form # 1 Form Filename: INPUTBOX.FRM
Version: 1.0 Date: July 5, 1992

Control object type	Default CtlName	Purpose
Command Button	Command1	Activates the countdown timer.
	Command2	Exits the test program.
Text Box	Text1	Displays the date and time.

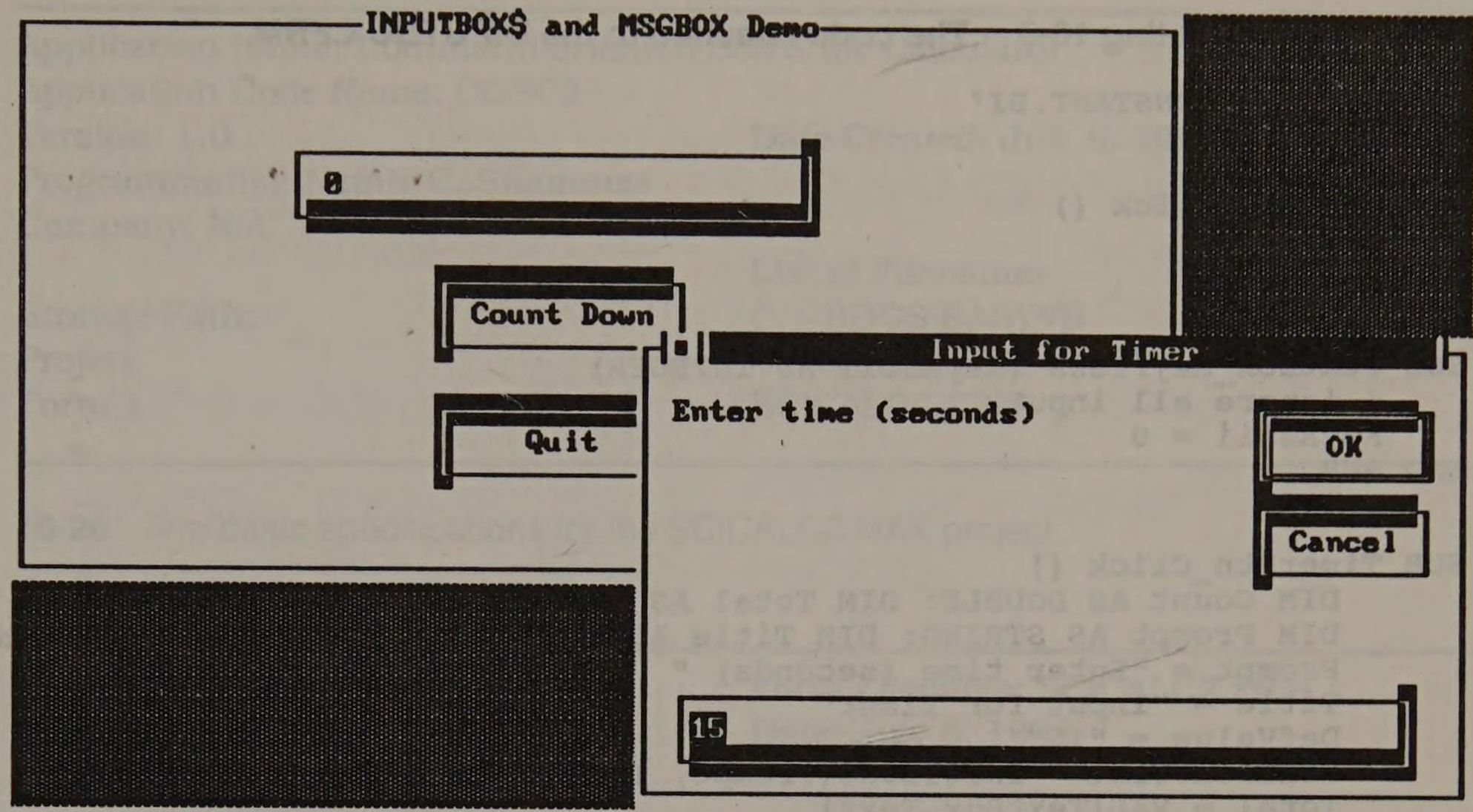
10-23 The general specifications for the INPUTBOX.FRM form.

Application (Code) Name: INPUTBOX
Form # 1
Version: 1.0 Date: July 5, 1992

Original control name	Property	New setting
Form	Caption	INPUTBOX$ and MSGBOX Demo
Command1	CtlName	TimerBtn
	Caption	Count Down
Command2	CtlName	QuitBtn
	Caption	Quit
Text1	CtlName	TextBox
	Caption	(empty string)

10-24 The new settings for the INPUTBOX.FRM form.

ments. The INPUTBOX$ statement displays a dialog box titled "Input for Timer" with an "Enter time (seconds)" prompt. The default time of 15 (seconds) appears in the edit text box. If you enter a zero or a negative value, the MSGBOX statement appears to inform you that your input was replaced with 15. The MSGBOX appears with the message "Timer input is adjusted to 15," and the title "For Your Information." The Information Message icon also appears to the left of the message text.



10-25 A sample session with the INPUTBOX program.

☞ Notice the code attached to the procedure TextBox_KeyPress. It contains a single statement that assigns zero to the KeyAscii argument. This statement prevents you from keying anything in the text box, making the text box an output-only control.

Run the above program and experiment with positive and negative timer input values. Observe the "For Your Information" message window appearance when you do not enter positive timer values.

Updating the scientific calculator

The Visual Basic test programs presented earlier were small programs that focused on a specific control object. Now that we have seen how these controls basically work, here is an updated version of the scientific calculator that uses option buttons, check boxes, and combo boxes.

The new version of the command-oriented calculator has the following new features:

- A drop-down list box replaces the Operator text box to offer you a list of the available operators and functions. Using this combo box you need not remember whether or not a function or operator is available.
- A check box used to indicate whether or not you want the variable names in the operand text boxes to be replaced with their values when you click on the Execute command button.

Listing 10-6 The code attached to the INPUTBOX.FRM.

```
'$INCLUDE: 'CONSTANT.BI'

SUB QuitBtn_Click ()
    END
END SUB

SUB TextBox_KeyPress (KeyAscii AS INTEGER)
    ' ignore all input
    KeyAscii = 0
END SUB

SUB TimerBtn_Click ()
    DIM Count AS DOUBLE: DIM Total AS DOUBLE: DIM Diff AS DOUBLE
    DIM Prompt AS STRING: DIM Title AS STRING: DIM DefValue AS STRING
    Prompt = "Enter time (seconds) "
    Title = "Input for Timer"
    DefValue = "15"
    TextBox.Text = INPUTBOX$(Prompt, Title, DefValue)
    Total = VAL(TextBox.Text)
    IF Total <= 0 THEN
        MSGBOX "Timer input is adjusted to 15", 64, "For Your Information"
        Total = 15
    END IF
    QuitBtn.Enabled = FALSE
    TimerBtn.Enabled = FALSE
    Count = TIMER
    DO WHILE (TIMER - Count) <= Total
        Diff = INT(Total - (TIMER - Count))
        TextBox.Text = STR$(Diff)
    LOOP
    BEEP
    TextBox.Text = STR$(0)
    TimerBtn.Enabled = TRUE
    QuitBtn.Enabled = TRUE
END SUB
```

- A check box assists you in automatically selecting the next variable name when you click on the Store button.
- A frame with three option buttons enables you to select the angle mode: radians, degrees, and gradients (grad for short). To convert from gradients to degrees, multiply the gradient angle by 0.9.

Figures 10-26 through 10-28 show the specifications for the SCICALC2.MAK project file and the SCICALC2.FRM form file. Figure 10-29 shows a sample session with this Visual Basic program.

Listing 10-7 contains the code attached to the SCICALC2.FRM and its controls. The listing shows event-handling procedures that are either new to this version or that have been modified from the previous version. The modified procedures are:

The StoreBtn_Click procedure Stores the result in the designated variable. The procedure adds statements that deal with selecting the next variable

Application Name: Command-oriented Scientific Calculator
Application Code Name: COSC2
Version: 1.0 Date Created: July 6, 1992
Programmer(s): Namir C. Shammas
Company: N/A

	List of Filenames
Storage Path:	C:\VBDOS\EASYVB
Project	SCICALC2.MAK
Form 1	SCICALC2.FRM

10-26 The basic specifications for the SCICALC2.MAK project.

Form # 1 Form Filename: SCICALC2.FRM
Version: 1.0 Date: July 6, 1992

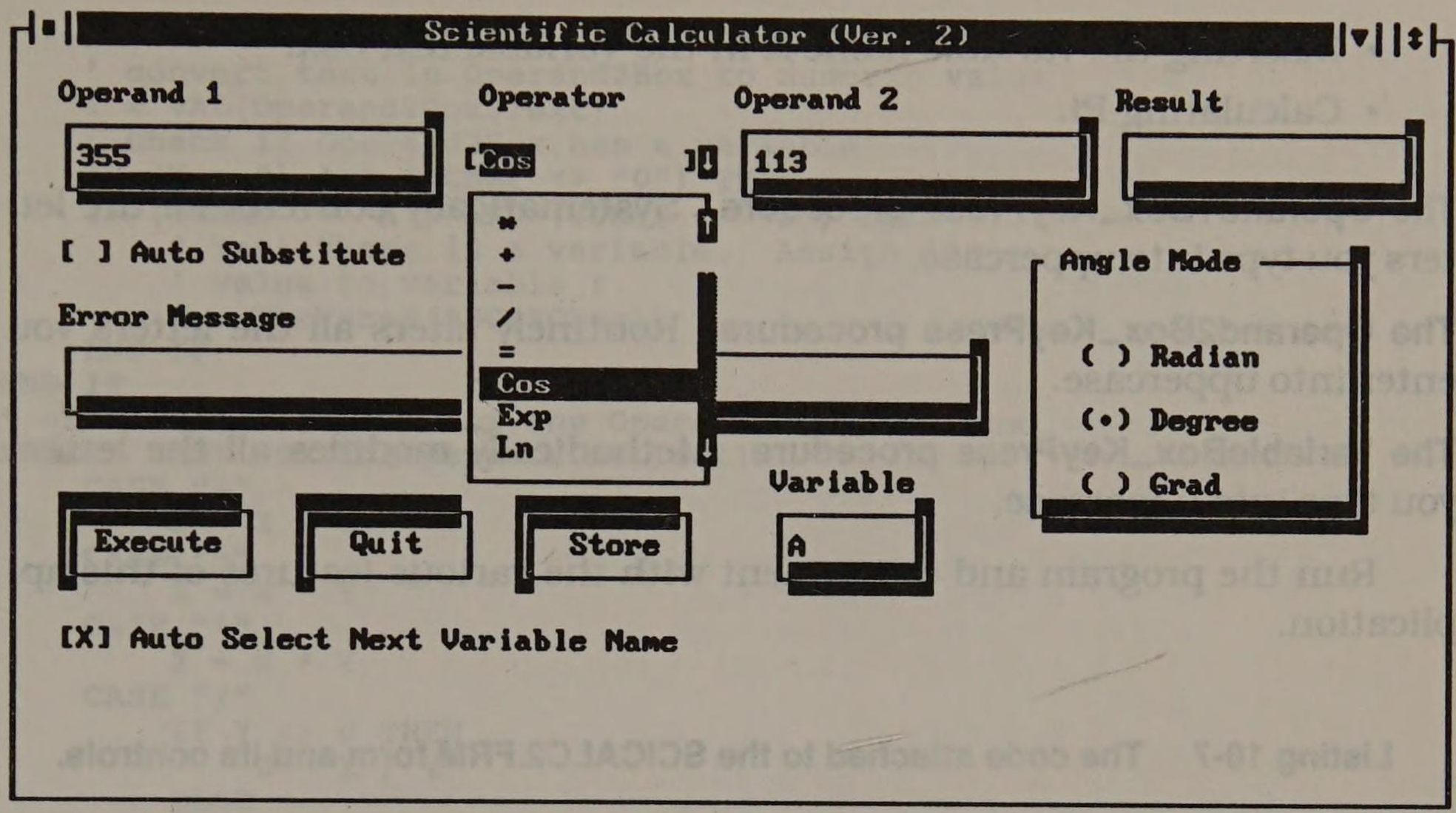
Control object type	Default CtlName	Purpose
Text Box	Text1	Key in operand 1.
	Text2	Key in operand 2.
	Text3	Display the result.
	Text4	Display error message.
	Text5	Key in the variable names.
Label	Label1	Label of Text1.
		Replace variable with numeric value.
	Label2	Label for Combo1 box.
	Label3	Label of Text2.
		Replace variable with numeric value.
	Label4	Label of Text3.
	Label5	Label of Text4.
	Label6	Clear error message.
		Label of Text5.
Command Button	Command1	Executes math operation.
	Command2	Exit the application.
	Command3	Stores the result in a variable.
Combo Box	Combo1	Offers the list of operators and functions.
Check Box	Check1	Automatically substitutes the variables in the operand boxes with their values.
	Check2	Automatically selects the next variable name when the Command3 button is clicked.
Frame Box	Frame1	Contains the angle option button.
Option Button	Option1	0 - Radians angle mode.
	Option2	1 - Degrees angle mode.
	Option3	2 - Gradient angle mode.

10-27 The list of controls in the SCICALC2.FRM form.

Application (Code) Name: COSC2
Form # 1
Version: 1.0 Date: July 6, 1992

Original control name	Property	New setting
Form1	Caption	Scientific Calculator (Ver. 2)
Text1	CtlName	Operand1Box
	Text	(empty string)
Text3	CtlName	Operand2Box
	Text	(empty string)
Text4	CtlName	ResultBox
	Text	(empty string)
Text5	CtlName	ErrorMessageBox
	Text	(empty string)
Text6	CtlName	VariableBox
	Text	(empty string)
Label1	CtlName	Operand1Lbl
	Caption	Operand 1
Label2	CtlName	OperatorLbl
	Caption	Operator
Label3	CtlName	Operator2Lbl
	Caption	Operand 2
Label4	CtlName	ResultLbl
	Caption	Result
Label5	CtlName	ErrorMessageLbl
	Caption	Error Message
Label6	CtlName	VariableLbl
	Caption	Variable
Command1	CtlName	ExecuteBtn
	Caption	Execute
Command2	CtlName	QuitBtn
	Caption	Quit
Command2	CtlName	StoreBtn
	Caption	Store
Combo1	CtlName	OperatorCmb
	Sorted	TRUE
	Style	2
Check1	CtlName	SubstituteChk
	Caption	Auto Substitute
Check2	CtlName	SelectChk
	Caption	Auto Select New Variable Name
Frame1	CtlName	AngleModeFrm
	Caption	Angle Mode
Option1	CtlName	RadianOpt
	Caption	Radian
Option1	CtlName	DegreeOpt
	Caption	Degree
Option1	CtlName	GradOpt
	Caption	Grad

10-28 The changes in the settings for the COSC2 application.



10-29 A sample session with the COSC2 project.

name. An If statement compares the SelectChk value property with 1. If the test is positive, the statements in the then clause change the value of the local variable Index (which stores the ASCII code for the current single-letter variable name) to select the next variable. These statements take into consideration the case when the current variable is Z. In this case, the value of Index is set to the function ASC("A"), the ASCII code for A.

The ExecuteBtn_Click procedure Executes the specified operation or function. The new version of this procedure obtains the name of the operation or function by examining the Text property of the OperatorCmb combo box. Another change is made in the procedure to deal with the various angle modes. The Case clauses for the sine, cosine, and tangent functions employ extended IF-THEN-ELSEIF statements to examine the Value property of the Radian, Degree, and Grad option buttons.

New procedures attached to the form and its controls are:

The Form_Load procedure Performs various initialization. They include the following:

- Setting the WindowState property to 2 (maximized). This causes the form to occupy the entire screen when the program starts running.
- Adding the operator and function names to the combo list. Notice that the items are added in unordered fashion. Because the Sorted property is set to TRUE, the list displays its items in alphabetical order.
- Setting the Value property of the Degree option button to TRUE.

- Inserting the variable name A in the Variable text box.
- Calculating Pi.

The Operand1Box_KeyPress procedure Systematically converts all the letters you type into uppercase.

The Operand2Box_KeyPress procedure Routinely alters all the letters you enter into uppercase.

The VariableBox_KeyPress procedure Methodically modifies all the letters you type into uppercase.

Run the program and experiment with the various features of this application.

Listing 10-7 The code attached to the SCICALC2.FRM form and its controls.

```
DECLARE SUB Operand1Lbl_Click ()
DECLARE SUB Operand2Lbl_Click ()
'$INCLUDE: 'CONSTANT.BI'
DIM SHARED TheVars(65 TO 90) AS DOUBLE
DIM SHARED Pi AS DOUBLE

SUB ErrorMessageLbl_Click ()
      ErrorMessageBox.Text = ""
END SUB

SUB ExecuteBtn_Click ()
    DIM X AS SINGLE: DIM Y  AS SINGLE: DIM Z  AS DOUBLE
    DIM Xchar AS STRING
    DIM YChar AS STRING
    ErrorMessageBox.Text = "" ' clear error message box
    ResultBox.Text = ""       ' clear result box
    ' is the Auto Substitute check box marked
    IF SubstituteChk.Value = 1 THEN
        Operand1Lbl_Click
        Operand2Lbl_Click
    END IF
    ' obtain the first character in the
    ' Operand1Box text box
    Xchar = LEFT$(Operand1Box.Text, 1)
    ' obtain the first character in the
    ' Operand2Box text box
    YChar = LEFT$(Operand2Box.Text, 1)
    ' convert string in Operand1Box text box to its
    ' numeric value.  Note that if the conversion fails,
    ' X will be assigned a zero.
    X = VAL(Operand1Box.Text)
    ' is X = 0 when Xchar is not "0"?  This test detects
    ' failed conversions
    IF (X = 0) AND (Xchar <> "0") THEN
        ' Yes!  Is Xchar in the range A to Z
        IF (Xchar >= "A") AND (Xchar <= "Z") THEN
            ' treat the content of Operand1Box as storing
            ' the name of a variable.  Assign the value of
            ' the variable to X
            X = TheVars#(ASC(Xchar))
        END IF
```

```
        END IF
        ' convert text in Operand2Box to numeric value
        Y = VAL(Operand2Box.Text)
        ' check if Operand2Box has a variable
        IF (Y = 0) AND (YChar <> "0") THEN
        IF (YChar >= "A") AND (YChar <= "Z") THEN
            ' Yes! there is a variable.  Assign its
            ' value to variable Y
            Y = TheVars#(ASC(YChar))
        END IF
    END IF
    ' examine the content of the OperatorBox text box
    SELECT CASE UCASE$(OperatorCmb.Text)
        CASE "+"
            Z = X + Y
        CASE "-"
            Z = X - Y
        CASE "*"
            Z = X * Y
        CASE "/"
            IF Y <> 0 THEN
                Z = X / Y
            ELSE
                ErrorMessageBox.Text = "Division by zero error"
                BEEP
            END IF
        CASE "^"
            Z = X ^ Y
        CASE "=" ' simply copy the value of the first
                 ' operand into the result box
            Z = X
        CASE "LN"
            IF X > 0 THEN
                Z = LOG(X)
            ELSE
                ErrorMessageBox.Text = "Bad function argument"
                BEEP
            END IF
        CASE "LOG"
            IF X > 0 THEN
                Z = LOG(X) / LOG(10)
            ELSE
                ErrorMessageBox.Text = "Bad function argument"
                BEEP
            END IF
        CASE "EXP"
            Z = EXP(X)
        CASE "SQR"
            IF X >= 0 THEN
                Z = SQR(X)
            ELSE
                ErrorMessageBox.Text = "Bad function argument"
                BEEP
            END IF
        CASE "SIN"
            IF RadianOpt.Value THEN
                Z = SIN(X)
            ELSEIF DegreeOpt.Value THEN
                Z = SIN(X * Pi / 180)
            ELSEIF GradOpt.Value THEN
                Z = SIN(X * .9 * Pi / 180)
            ELSE
                Z = SIN(X)
            END IF
```

Listing 10-7 Continued.

```
CASE "COS"
    IF RadianOpt.Value THEN
        Z = COS(X)
    ELSEIF DegreeOpt.Value THEN
        Z = COS(X * Pi / 180)
    ELSEIF GradOpt.Value THEN
        Z = COS(X * .9 * Pi / 180)
    ELSE
        Z = COS(X)
    END IF
CASE "TAN"
    IF RadianOpt.Value THEN
        Z = TAN(X)
    ELSEIF DegreeOpt.Value THEN
        Z = TAN(X * Pi / 180)
    ELSEIF GradOpt.Value THEN
        Z = TAN(X * .9 * Pi / 180)
    ELSE
        Z = TAN(X)
    END IF
  CASE ELSE
    ErrorMessageBox.Text = "Invalid operator"
    BEEP
END SELECT
' if the error message box is still empty, there is
' no error.  Therefore, convert the variable Z into
' a string and display it in ResultBox
IF ErrorMessageBox.Text = "" THEN
    ResultBox.Text = STR$(Z)
END IF
END SUB

SUB Form_Load ()
    WindowState = 2 ' set window to maximized state
    ' insert list of operators and functions in the
    ' Operator combo box
    OperatorCmb.ADDITEM "+"
    OperatorCmb.ADDITEM "-"
    OperatorCmb.ADDITEM "*"
    OperatorCmb.ADDITEM "/"
    OperatorCmb.ADDITEM "^"
    OperatorCmb.ADDITEM "="
    OperatorCmb.ADDITEM "Sqr"
    OperatorCmb.ADDITEM "Tan"
    OperatorCmb.ADDITEM "Cos"
    OperatorCmb.ADDITEM "Exp"
    OperatorCmb.ADDITEM "Ln"
    OperatorCmb.ADDITEM "Log"
    OperatorCmb.ADDITEM "Sin"
    ' select the Dgree option box
    DegreeOpt.Value = TRUE
    ' insert the variable A into the variable box
    VariableBox.Text = "A"
    ' calculate Pi
    Pi = 4 * ATN(1)
END SUB

SUB Operand1Box_KeyPress (KeyAscii AS INTEGER)
    KeyAscii = ASC(UCASE$(CHR$(KeyAscii)))
END SUB
```

```
SUB Operand1Lbl_Click ()
    DIM Xchar AS STRING
    ' get the first character in the Operand1Box
    Xchar = LEFT$(Operand1Box.Text, 1)
    ' is the character in the range A to Z?
    IF (Xchar >= "A") AND (Xchar <= "Z") THEN
        ' Yes!  Obtain the variable from the array and write
        ' its numeric value to the text box
        Operand1Box.Text = STR$(TheVars#(ASC(Xchar)))
    END IF
END SUB

SUB Operand2Box_KeyPress (KeyAscii AS INTEGER)
    KeyAscii = ASC(UCASE$(CHR$(KeyAscii)))
END SUB

SUB Operand2Lbl_Click ()
    DIM Xchar AS STRING
    ' get the first character in the
    ' Operand2Box
    Xchar = LEFT$(Operand2Box.Text, 1)
    ' is the character in the range A to Z?
    IF (Xchar >= "A") AND (Xchar <= "Z") THEN
        ' Yes!  Obtain the variable from the array and write
        ' its numeric value to the text box
        Operand2Box.Text = STR$(TheVars#(ASC(Xchar)))
    END IF

END SUB

SUB QuitBtn_Click ()
    END
END SUB

SUB StoreBtn_Click ()
    DIM C AS STRING
    DIM Index AS INTEGER
    ' process the contents of the Variable text box
    IF VariableBox.Text <> "" THEN
        ' obtain the uppercase of the first character in the text box
        C = UCASE$(LEFT$(VariableBox.Text, 1))
        ' is the character in the range A to Z?
        IF (C >= "A") AND (C <= "Z") THEN
            ' Yes! Get the ASCII code for the character
            Index = ASC(C)
            ' store the value of the Result box in the array TheVars#
            TheVars#(Index) = VAL(ResultBox.Text)
            ' clear the error message box
            ErrorMessageBox.Text = ""
            ' select the next variable
            IF SelectChk.Value = 1 THEN
                IF Index = ASC("Z") THEN
                    Index = ASC("A")
                ELSE
                    Index = Index + 1
                END IF
                VariableBox.Text = CHR$(Index)
            END IF
        ELSE
            ' No! The text box has an invalid variable name
            ErrorMessageBox.Text = "Invalid variable name"
            BEEP
        END IF
    ELSE
```

Listing 10-7 Continued.

```
        ' the text box is empty!
        ErrorMessageBox.Text = "Missing variable name"
        BEEP
    END IF
END SUB

SUB VariableBox_KeyPress (KeyAscii AS INTEGER)
    KeyAscii = ASC(UCASE$(CHR$(KeyAscii)))
END SUB
```

11
Output

The last chapter discussed various Visual Basic input controls. This chapter looks at the various ways to output both formatted and unformatted information. A few of these methods have already been discussed. The method for using the text box for output was presented in chapter 7, with a focus on using text boxes to simulate pseudoscreens. In the last chapter I also presented the MsgBox statement (while discussing the very similar MsgBox function) that displays information. This chapter covers:

- Using the Visual Basic PRINT statement.
- Formatting output.
- Sending output to the printer.

Output to a form, picture box, and the printer

Visual Basic supports the PRINT method to display text in a form or in a picture box. In addition, the PRINT method can send output to the line printer connected to your system. There are properties common to all three objects (that is, the form, the picture box, and the printer) and there are properties common to the form and picture box; and form and printer.

The main difference between the output to the printer and the other two forms is that the print position of a printer is always advancing. By contrast, the print position of a form or picture box is more flexible. You can reposition that print position to overwrite previous lines. This means that the form and the picture forms act as miniature screens. The following properties and methods can be used to manipulate the text location in a form or picture box:

The CLS method Clears the text emitted by a PRINT statement in form or picture box. The general syntax is:

[form | pictureBox].CLS

The controls drawn on a form are not affected.

The CurrentX and CurrentY properties Return the (X, Y) location of the current print position. The general syntax for using these properties is:

[form.] [pictureBox.] CurrentX [=X%]
[form.] [pictureBox.] CurrentY [=Y%]

☞ When the CLS method is used, the CurrentX and CurrentY properties are both set to 0. This means that the upper-left corner of a form or picture box is (0, 0) AND NOT (1, 1).

The TextHeight method Returns the height in lines of text string. The general syntax for the TextHeight method is:

[form.] [pictureBox.] TextHeight(stringExpression$)

The TextHeight is expressed in terms of the character length.

The TextWidth method Returns the width of a string text. The general syntax for this method is shown next:

[form.] [pictureBox.] TextWidth(stringExpression$)

The Visual Basic PRINT statement

The Visual Basic PRINT statement and syntax is basically the same one implemented in QuickBasic, BASICA, and GW-Basic. The general syntax for the PRINT method is:

{ [form.] [pictureBox.] Printer.} PRINT [USING f$] [exprList] [{;|,}]

The PRINT statement can print blank lines (in the absence of an expression list), the value of single expression, or a list of values. The comma enables you to use standard tabbing, while the colon suppresses the carriage return after a value is emitted. As with QuickBasic and the other Basics, you can use the TAB function to set your own tabbing. Moreover, you can use the SPC function to specify the spacing between two printed items. The PRINT method also supports the USING clause for formatted output, just like with QuickBasic and GW-BASIC. Examples of using the PRINT method are shown below:

```
MyPicture.PRINT
Printer.PRINT
Printer.PRINT 355 / 113 ' prints the value of pi
PRINT USING "##.# ##.## #.####"; X, Y, Z
MyPicture.PRINT "X = "; X, "Y = "; Y
Printer.PRINT Date$; Tab(65); Time$
```

The FORMAT$ function

In addition to the USING clause of the PRINT method, Visual Basic offers the FORMAT$ function to return a formatted string. The general syntax of the FORMAT$ function is:

```
FORMAT$(expression [,formatString$])
```

The expression can be numeric or string. The optional formatString$ contains the format string. The syntax of the format string is the same one used in the USING clause. When the formatString$ is an empty string, the FORMAT$ function works like the STR$ function.

Printer output

The printer object has the following properties and methods that assist in managing the printer output:

The NEWPAGE method Performs an explicit form feed. The printer object automatically sends a form feed when you have printed enough lines to fill a page.

The ENDDOC method Releases the print device or print spooler. If the ENDDOC is called right after a NEWPAGE, no additional form feed occurs. Otherwise, the printer advances the current page that contains the last part of the document or output you are printing.

The PrintTarget property Allows you to select the printer or output filename. This property is valuable in directing the output. The general syntax for the PrintTarget property is:

```
PRINTER.PrintTarget = target$
```

Where target is either a filename or a valid printer port such as "LPT1," "LPT2," and "LPT3."

You can print a form using the form.PRINTFORM method. This method prints the entire form, regardless of how visible the form is. To print the graphics in a form you must set the AutoRedraw property to TRUE. The quality of the graphics output on your printer depends on the printer.

A Pseudoscreen in a picture box

Chapter 7 presented the PSCREEN.MAK project that uses a multiline text to emulate a pseudoscreen. This section presents another version of the pseudoscreen that uses PRINT statements in a picture box. Using the PRINT statement in concert with the CurrentX and CurrentY properties allows the program to display text anywhere in the picture box. This output method makes using an array of strings less critical. The code implementation in Listing 11-1 uses no string arrays to store a copy of the screen text. As a result, the features of scrolling, storing, and recalling a screen are missing

from this new version. I leave it as an exercise to the reader to implement such a version. Without using string arrays, the picture-based pseudoscreen uses less memory than the text box version.

Figures 11-1 through 11-3 show the specifications for the project and form files. Figure 11-4 shows a sample session with the PSCREEN2.MAK project.

Application Name: Pseudoscreen Test Program
Application Code Name: PSCREEN2
Version: 1.0 Date Created: July 8, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	PSCREEN2.MAK
Form 1	PSCREEN2.FRM

11-1 General project specifications for the PSCREEN2 project.

Form # 1 Form Filename: PSCREEN2.FRM
Version: 1.0 Date: July 8, 1992

Control object type	Default CtlName	Purpose
Command Button	Command1	Exits the test program.
	Command2	Clears the pseudoscreen.
	Command3	Moves the cursor.
	Command4	Locates the cursor.
	Command5	Writes the text in Text4 to the pseudoscreen (Text1).
	Command6	Moves the cursor to the next line.
Text Box	Text1	The cursor column number.
	Text2	The cursor row number.
	Text3	Text to write on the pseudoscreen.
Label	Label1	Label for Text1.
	Label2	Label for Text2.
	Label3	Label for Text3.
Picture Box	Picture1	The pseudoscreen.

11-2 List of control objects for the PSCREEN2 project.

Listing 11-1 shows the code attached to the PSCREEN2.FRM form. The new version of the pseudoscreen program differs from the first one in the following ways:

- The general declarations section has fewer declarations. While the constants MAX_ROWS and MAX_COLS are still declared, they have smaller values than their counterparts in PSCREEN.FRM. The string

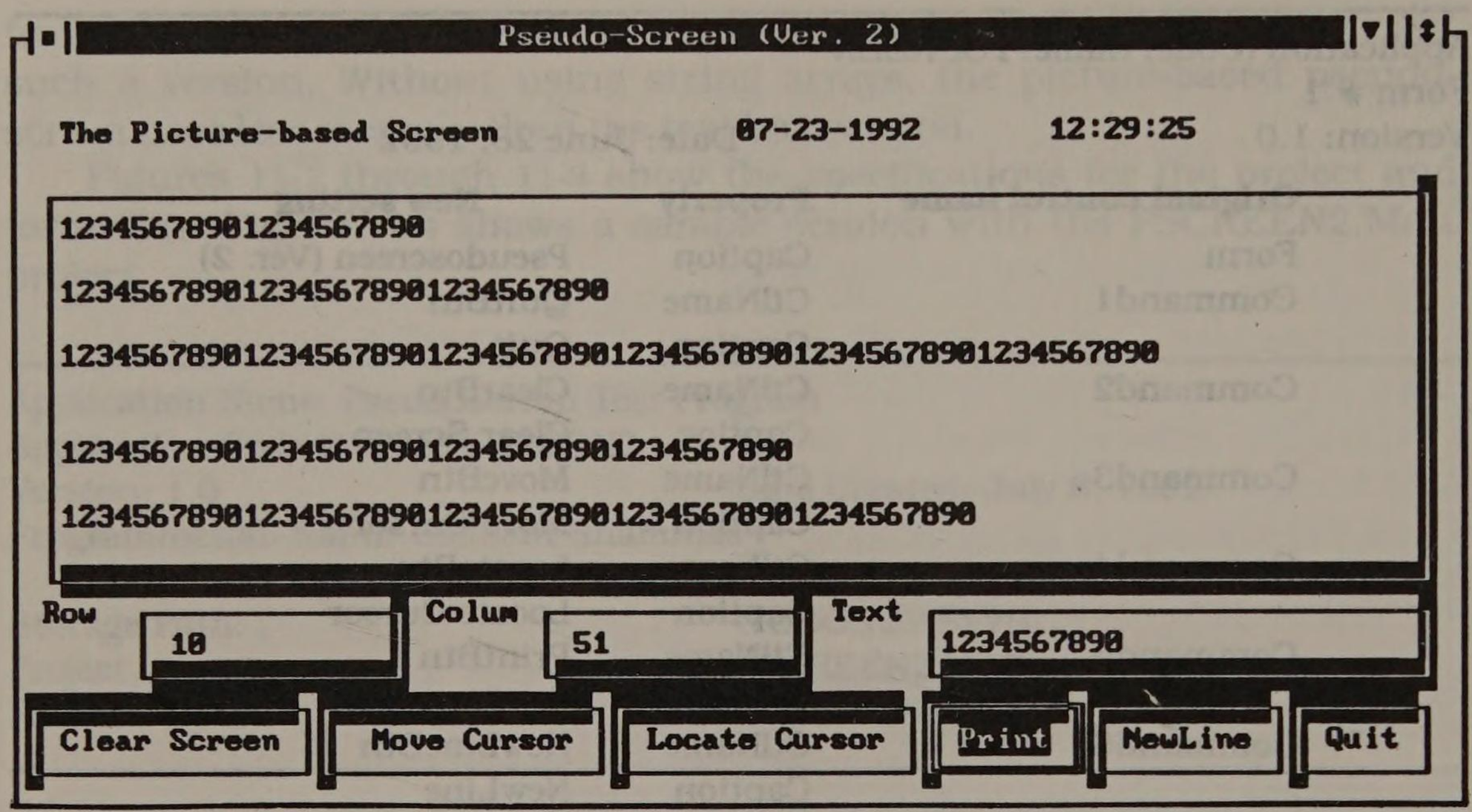
Application (Code) Name: PSCREEN
Form # 1
Version: 1.0 Date: June 28, 1992

Original control name	Property	New setting
Form	Caption	Pseudoscreen (Ver. 2)
Command1	CtlName	QuitBtn
	Caption	Quit
Command2	CtlName	ClearBtn
	Caption	Clear Screen
Command3	CtlName	MoveBtn
	Caption	Move Cursor
Command4	CtlName	LocateBtn
	Caption	Locate Cursor
Command5	CtlName	PrintBtn
	Caption	Print
Command6	CtlName	NewLineBtn
	Caption	NewLine
Text1	CtlName	CursorXBox
	Text	(empty string)
Text2	CtlName	CursorYBox
	Text	(empty string)
Text3	CtlName	TextBox
	Text	(empty string)
Label1	CtlName	ColumnLbl
	Caption	Column
Label2	CtlName	RowLbl
	Caption	Row
Label3	CtlName	TextLbl
	Caption	Text
Picture1	CtlName	ScreenPct

11-3 The modified control settings for the PSCREEN project.

arrays and the cursor management variables are replaced with the ScaleX and ScaleY variables that store the values for TextHeight("X") and TextWidth("Y"). I am using these variables for the sake of demonstration the latter two functions. You can remove the ScaleX and ScaleY variables, because they actually store 1.

- The event-handling procedures for scrolling, saving, and recalling a screen are no longer implemented, because their respective features are not offered by the new program version.
- The auxiliary procedures ClearScreen, GotoXY, NewLine, WhereX, and WhereY are still present, but with a much simpler and shorter code. Cursor management now uses the properties CurrentX and CurrentY and the ScaleX and ScaleY variables.



11-4 A sample session with the PSCREEN2 program.

- The PPrint procedure in PSCREEN.FRM has been completely removed and its calls are substituted with the Visual Basic PRINT method.
- The LocateBtn_Click procedure now includes statements to clear the form's own text and print a pseudolabel for the pseudoscreen that includes the current date and time.
- The new program version has the Form_Load procedure that initializes the application. This includes the following:
  - ~ Maximizing the window at run time.
  - ~ Assigning values to the ScaleX and ScaleY variables.
  - ~ Clearing the pseudoscreen by calling ClearBtn_Click.
  - ~ Assigning the string "1234567890" to the Text property of TextBox.

Run the PSCREEN2.FRM program. Click on the Print and New Line command buttons and watch the text being displayed in various parts of the pseudoscreen. As you click the Print and New Line buttons, observe the cursor text boxes. Experiment with assigning new cursor location and then press the Print button. When you are done with the program, click the Quit command button.

The Calendar form

This section illustrates the use of an additional form that specializes in displayed PRINT statement output data. Printing to the main form, which nor-

Listing 11-1 The code attached to the PSCREEN2.FRM form.

```
DECLARE SUB ClearScreen ()
DECLARE SUB GotoXY (X AS INTEGER, Y AS INTEGER)
DECLARE FUNCTION WhereX () AS INTEGER
DECLARE FUNCTION WhereY () AS INTEGER
DECLARE SUB NewLine ()
DECLARE SUB ClearBtn_Click ()
DECLARE SUB LocateBtn_Click ()
' constants that define the size of the emulated screen
CONST MAX_ROWS = 24, MAX_COLS = 75

DIM SHARED ScaleX AS SINGLE: DIM SHARED ScaleY  AS SINGLE


SUB ClearBtn_Click ()
    ClearScreen
    LocateBtn_Click
END SUB

SUB ClearScreen ()
    ScreenPct.CLS
END SUB

SUB Form_Load ()
    WindowState = 2
    ScaleX = ScreenPct.TEXTWIDTH("X")
    ScaleY = ScreenPct.TEXTHEIGHT("X")
    ClearBtn_Click
    ' set initial test string
    TextBox.Text = "1234567890"
END SUB

SUB GotoXY (X AS INTEGER, Y AS INTEGER)
    ' move the hidden cursor to (X,Y)
    IF (X < 1) OR (Y < 1) THEN EXIT SUB
    IF (Y > MAX_ROWS) OR (X > MAX_COLS) THEN EXIT SUB
    ScreenPct.CurrentX = ScaleX * (X - 1) + .5
    ScreenPct.CurrentY = ScaleY * (Y - 1) + .5
END SUB

SUB LocateBtn_Click ()
    CursorXBox.Text = STR$(WhereX())
    CursorYBox.Text = STR$(WhereY())
    CLS
    PRINT :  PRINT
    PRINT SPC(2); "The Picture-based Screen";
    PRINT TAB(40); DATE$, TIME$
END SUB

SUB MoveBtn_Click ()
```

Listing 11-1 Continued.

```
    DIM X AS INTEGER
    DIM Y AS INTEGER
    X = VAL(CursorXBox.Text)
    Y = VAL(CursorYBox.Text)
    GotoXY X, Y
    LocateBtn_Click
END SUB

SUB NewLine ()
    IF WhereY() < MAX_ROWS THEN
        ScreenPct.CurrentY = ScreenPct.CurrentY + ScaleY
        ScreenPct.CurrentX = 0
    END IF
END SUB

SUB NewLineBtn_Click ()
    NewLine
    LocateBtn_Click
END SUB

SUB PrintBtn_Click ()
    IF (WhereX() + LEN(TextBox.Text)) < MAX_COLS THEN
        ScreenPct.PRINT TextBox.Text;
        LocateBtn_Click
    END IF
END SUB

SUB QuitBtn_Click ()
        END
END SUB

FUNCTION WhereX () AS INTEGER
    DIM X AS DOUBLE
    ' return the value of CurrentX
    X = 1 + ScreenPct.CurrentX / ScaleX
    IF X < 1 THEN X = X + 1
    WhereX = X
END FUNCTION

FUNCTION WhereY () AS INTEGER
    DIM Y AS DOUBLE
    ' return the value of CurrentY
    Y = 1 + ScreenPct.CurrentY / ScaleY
    IF Y < 1 THEN Y = Y + 1
    WhereY = Y
END FUNCTION
```

mally contains various controls, is cumbersome. By contrast, using a blank form (called the output form) makes the job of sending output to a form much easier, because there is little or nothing in the way. You will need at least one control in the output form. This control can be either a command button or better yet a menu, because menus stay out of the way.

The example presented in this section is a simple calendar maker. Specify the year and month number in text boxes located in the main form and then click a command button to view the calendar in an output form. The output form has a menu with Print and Exit selections. There are no menu options. Simply click the Print menu selection to print the current form. Click on the Exit selection to return to the main form.

Figures 11-5 through 11-9 contain the specifications for the project file CALEND1.MAK and the form files CALEND1.FRM and CAL1FRM.FRM. Figure 11-10 shows a sample session with the calendar maker application.

Application Name: Simple Calendar Maker
Application Code Name: CALEND1
Version: 1.0 Date Created: July 8, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	CALEND1.MAK
Form 1	CALEND1.FRM
Form 2	CAL1FRM.FRM

11-5 The basic specifications for the CALEND1.MAK project.

Listing 11-2 shows the following procedures attached to the CALEND1.FRM:

The YearBox_KeyPress and MonthBox_KeyPress procedures Allow only the digits and the backspace character. All other characters are filtered out.

The QuitBtn_Click procedure Exits the application using an End statement.

The MakeBtn_Click procedure This procedure is the work horse of the calendar maker; it performs the following tasks:

- Declares local constants for the left margin and spacing between days.
- Sets the AutoRedraw property of the output form to TRUE. This enables you to minimize the output form and then maximize it without losing the form contents.
- Sets the format for the day number.
- Selects the current year if the Year text box is empty.

Form # 1 Form Filename: CALEND1.FRM
Version: 1.0 Date: July 8, 1992

Control object type	Default CtlName	Purpose
Command Button	Command1	Makes and displays a calendar.
	Command2	Exits the application.
Text Box	Text1	Stores the year number.
	Text2	Stores the month number.
Label	Label1	Label for the year number.
	Label2	Label for the month number.

11-6 The default controls for the CALEND1.FRM form.

Form # 2 Form Filename: CAL1FRM.FRM
Version: 1.0 Date: July 8, 1992

Caption	Property	Setting	Purpose
&Print	CtlName	PrintCom	Prints form.
E&xit	CtlName	ExitCom	Returns to the main form.

11-7 The menu structure for the CAL1FRM.FRM form.

Application (Code) Name: CALEND1
Form # 1
Version: 1.0 Date: July 8, 1992

Original control name	Property	New setting
Form	Caption	Calendar Maker
	FormName	CalendarMakerFrm
Command1	CtlName	MakeCalendarBtn
	Caption	MakeCalendar
	Default	TRUE
Command2	CtlName	QuitBtn
	Caption	Quit
	Cancel	TRUE
Text1	CtlName	YearBox
	Text	(empty string)
Text2	CtlName	MonthBox
	Text	(empty string)
Label1	CtlName	YearLbl
	Caption	Year Number
Label2	CtlName	MonthLbl
	Caption	Month Number

11-8 The updated settings of the CALEND1.FRM form.

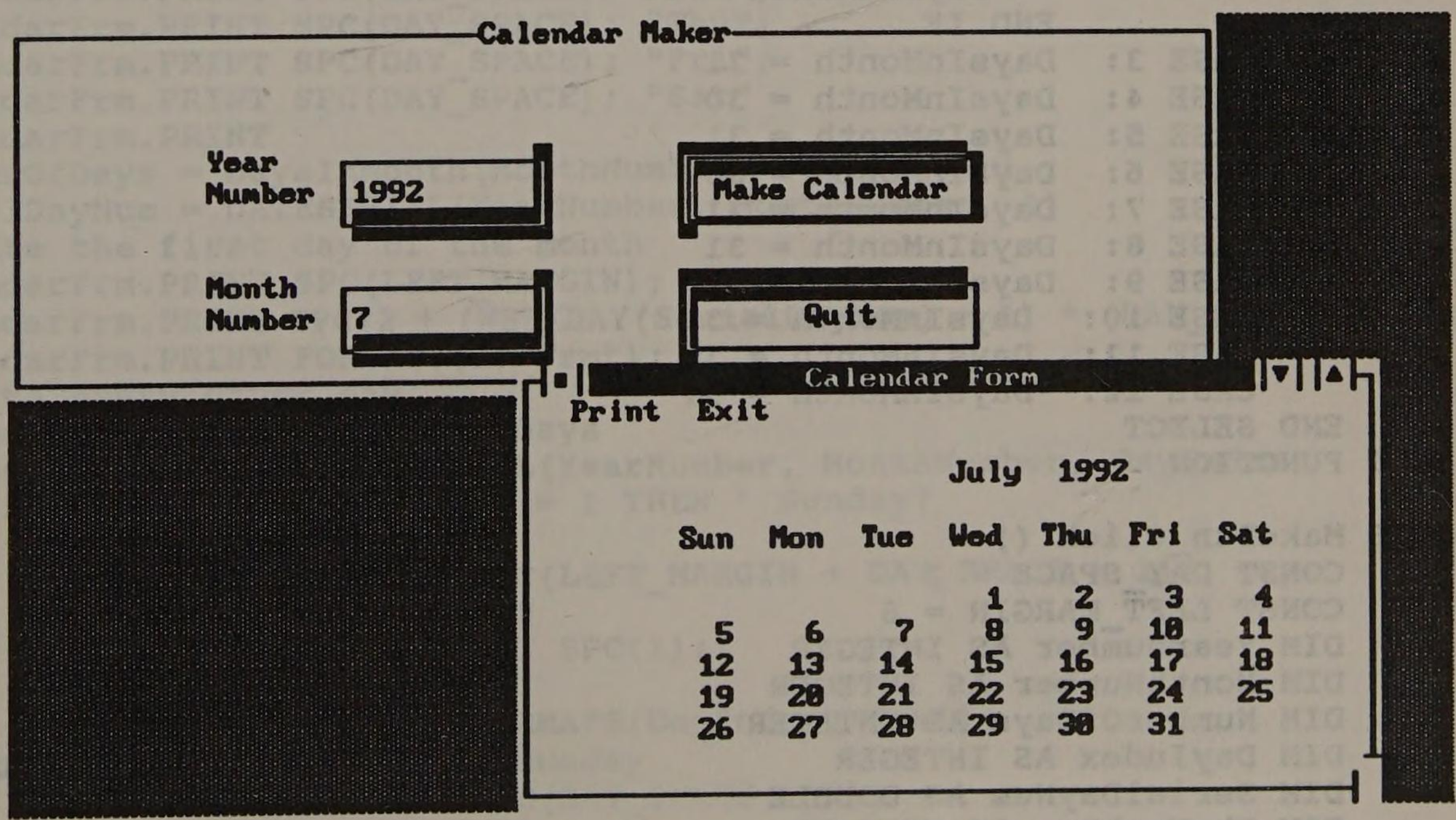
Application (Code) Name: CALEND1
Form # 2
Version: 1.0 Date: July 8, 1992

Original control name	Property	New setting
Form	Caption	Calendar Form
	FormName	CalendarFrm

11-9 The updated settings of the CAL1FRM.FRM form.

- Selects the current month if the Month text box is empty.
- Obtains the year and month numbers.
- Checks if either the year or the month numbers are invalid. If either one is not valid, a message box displays an error message and the procedure exits.
- Applies a series of PRINT statements to write the calendar month name, year, weekdays, and the location of the month days. The SPC function is used to place the day numbers in their proper location. The FORMAT$ function is utilized to obtain a formatted day number. A FOR-NEXT loop writes the days of the month under the appropriate week day, starting with the second day of the month—the first day of the month requires slightly different code.

The procedure uses date/time procedures and functions, such as, the Now, DateSerial, and WeekDay functions.



11-10 A sample session with the CALEND1 program.

The MonthName procedure Returns the name of a month.

The DaysInMonth procedure Returns the number of days in a month, taking into consideration leap years.

Listing 11-3 shows the two event-handling procedures attached to the menu selections of the output form.

Run the program by pressing the F5 function key. To view the calendar of the current month, simply click the Make Calendar command button. Experiment with minimizing and then maximizing the output form. The calendar data are preserved during this resizing test. Click the Print menu or press the ALT-P keys to print the output form. Click the Exit menu or press the ALT-X keys to exit the output form. Experiment by typing in other months and years.

Listing 11-2 The code attached to the CALEND1.FRM.

```
'$FORM CalendarFrm
DECLARE FUNCTION MonthName (MonthNumber AS INTEGER) AS STRING
DECLARE FUNCTION DaysInMonth (MonthNum AS INTEGER, YearNum AS INTEGER) AS
INTEGER
'$INCLUDE: 'CONSTANT.BI'

FUNCTION DaysInMonth (MonthNum AS INTEGER, YearNum AS INTEGER) AS INTEGER
    SELECT CASE MonthNum
    CASE 1:  DaysInMonth = 31
    CASE 2
             IF ((YearNum - 1988) MOD 4) = 0 THEN
                DaysInMonth = 29
             ELSE
                DaysInMonth = 28
             END IF
    CASE 3:  DaysInMonth = 31
    CASE 4:  DaysInMonth = 30
    CASE 5:  DaysInMonth = 31
    CASE 6:  DaysInMonth = 30
    CASE 7:  DaysInMonth = 31
    CASE 8:  DaysInMonth = 31
    CASE 9:  DaysInMonth = 30
    CASE 10:  DaysInMonth = 31
    CASE 11:  DaysInMonth = 30
    CASE 12:  DaysInMonth = 31
END SELECT
END FUNCTION

SUB MakeBtn_Click ()
    CONST DAY_SPACE = 2
    CONST LEFT_MARGIN = 8
    DIM YearNumber AS INTEGER
    DIM MonthNumber AS INTEGER
    DIM NumberOfDays AS INTEGER
    DIM DayIndex AS INTEGER
    DIM SerialDayNum AS DOUBLE
    DIM TheWeekDay AS INTEGER
```

```
DIM Frmt AS STRING
CalendarFrm.AutoRedraw = TRUE
Frmt = "###" ' set page format
' if Year text box is empty use the current year
IF YearBox.Text = "" THEN
    YearBox.Text = FORMAT$(YEAR(NOW))
END IF
' if the Month text box is empty use the current month
IF MonthBox.Text = "" THEN
    MonthBox.Text = FORMAT$(MONTH(NOW))
END IF
YearNumber = VAL(YearBox.Text) ' get the year number
MonthNumber = VAL(MonthBox.Text) ' get the month number
' is the year number invalid?
IF YearNumber < 1980 OR YearNumber > 2099 THEN
    MSGBOX "Invalid Year number", 0, "Input Error"
    EXIT SUB
END IF
' is the month number invalid?
IF MonthNumber < 1 OR MonthNumber > 12 THEN
    MSGBOX "Invalid Month number", 0, "Input Error"
    EXIT SUB
END IF
' show output form
CalendarFrm.SHOW
' clear the form and build a new calendar
CalendarFrm.CLS
CalendarFrm.PRINT
CalendarFrm.PRINT SPC(LEFT_MARGIN);
CalendarFrm.PRINT "                    "; MonthName(MonthNumber);
CalendarFrm.PRINT " "; YearNumber
CalendarFrm.PRINT
CalendarFrm.PRINT SPC(LEFT_MARGIN); "Sun";
CalendarFrm.PRINT SPC(DAY_SPACE); "Mon";
CalendarFrm.PRINT SPC(DAY_SPACE); "Tue";
CalendarFrm.PRINT SPC(DAY_SPACE); "Wed";
CalendarFrm.PRINT SPC(DAY_SPACE); "Thu";
CalendarFrm.PRINT SPC(DAY_SPACE); "Fri";
CalendarFrm.PRINT SPC(DAY_SPACE); "Sat"
CalendarFrm.PRINT
NumberOfDays = DaysInMonth(MonthNumber, YearNumber)
SerialDayNum = DATESERIAL(YearNumber, MonthNumber, 1)
' write the first day of the month
CalendarFrm.PRINT SPC(LEFT_MARGIN);
CalendarFrm.PRINT SPC(2 + (WEEKDAY(SerialDayNum) - 1) * (DAY_SPACE + 3));
CalendarFrm.PRINT FORMAT$(1, Frmt);
' write every other day
FOR DayIndex = 2 TO NumberOfDays
    SerialDayNum = DATESERIAL(YearNumber, MonthNumber, DayIndex)
    IF WEEKDAY(SerialDayNum) = 1 THEN ' Sunday?
        CalendarFrm.PRINT
        CalendarFrm.PRINT SPC(LEFT_MARGIN + DAY_SPACE - 1);
        IF DayIndex < 10 THEN
            CalendarFrm.PRINT SPC(1);
        END IF
        CalendarFrm.PRINT FORMAT$(DayIndex, Frmt);
    ELSE ' a day other than Sunday
        CalendarFrm.PRINT SPC(DAY_SPACE + 1);
        IF DayIndex < 10 THEN
```

Listing 11-2 Continued.

```
                CalendarFrm.PRINT SPC(1);
            END IF
            CalendarFrm.PRINT FORMAT$(DayIndex, Frmt);
        END IF
    NEXT DayIndex
END SUB

SUB MonthBox_KeyPress (KeyAscii AS INTEGER)
    IF KeyAscii = 8 THEN EXIT SUB
    IF KeyAscii < ASC("0") OR KeyAscii > ASC("9") THEN
        KeyAscii = 0
    END IF
END SUB

FUNCTION MonthName (MonthNumber AS INTEGER) AS STRING
    SELECT CASE MonthNumber
        CASE 1:  MonthName = "January"
        CASE 2:  MonthName = "February"
        CASE 3:  MonthName = "March"
        CASE 4:  MonthName = "April"
        CASE 5:  MonthName = "May"
        CASE 6:  MonthName = "June"
        CASE 7:  MonthName = "July"
        CASE 8:  MonthName = "August"
        CASE 9:  MonthName = "September"
        CASE 10:  MonthName = "October"
        CASE 11:  MonthName = "November"
        CASE 12:  MonthName = "December"
    END SELECT
END FUNCTION

SUB QuitBtn_Click ()
    END
END SUB

SUB YearBox_KeyPress (KeyAscii AS INTEGER)
    IF KeyAscii = 8 THEN EXIT SUB
    IF KeyAscii < ASC("0") OR KeyAscii > ASC("9") THEN
        KeyAscii = 0
    END IF
END SUB
```

Listing 11-3 The code attached to the output form CAL1FRM.FRM.

```
'$INCLUDE: 'CONSTANT.BI'

SUB ExitCom_Click ()
    HIDE
END SUB

SUB PrintCom_Click ()
    PRINTFORM
END SUB
```

The Text File printer

The last example in this chapter deals with sending text to the printer. The application is a simple text file printer. The program has a text box that stores the name of the single text file you want to print. A Print command button produces a paginated printer output that includes the page number, the current date, and the current time.

Figures 11-11 through 11-13 show the specifications for the project file PRINTFL1.MAK and the form file PRINTFL1.FRM. Figure 11-14 shows a sample session with the text file printing program.

Listing 11-4 contains the code attached to the PRINTFL1.FRM form. The general declarations section declares the constant MAX_LINES to specify the maximum number of lines per page (including the heading). The variables Filename and CurrentLineNumber store the name of the printed file and the current line number, respectively. These variables are used in multiple procedures. The variable Page stores the page number.

The relevant procedures in the PRINTFL1.FRM form are:

The PrintHeading procedure Prints the filename, page number, and the current date and time. The procedure resets the value of the CurrentLineNumber variable to 4, the number of lines printed by the procedure itself.

Application Name: Text File Printer (Version 1)
Application Code Name: PRINTFL1
Version: 1.0 Date Created: July 8, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	PRINTFL1.MAK
Form 1	PRINTFL1.FRM

11-11 The basic specifications for the PRINTFL1.MAK project.

Form # 1 Form Filename: PRINTFL1.FRM
Version: 1.0 Date: July 8, 1992

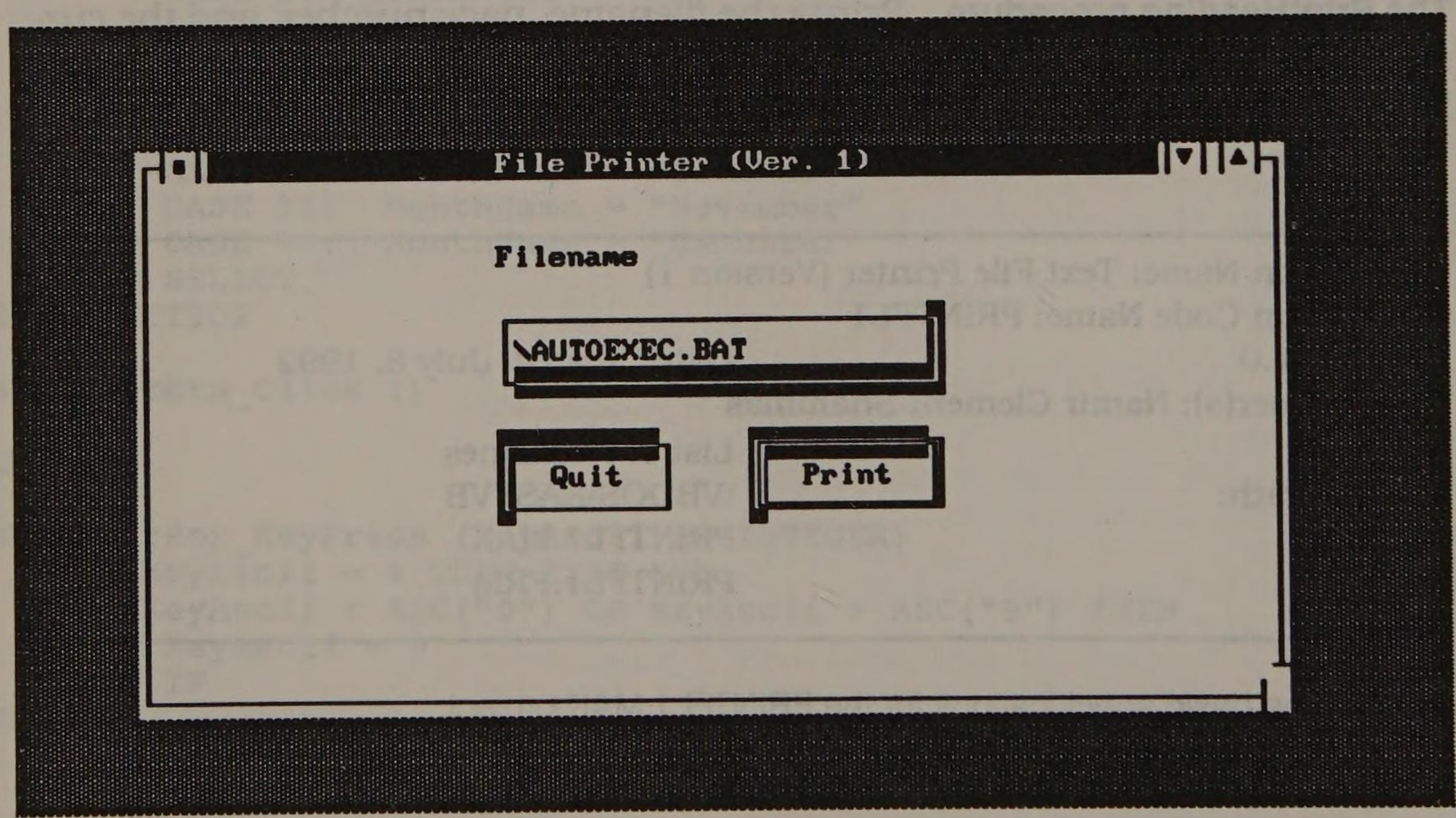
Control object type	Default CtlName	Purpose
Command Button	Command1	Prints a text file specified in Text1.
	Command2	Exits the application.
Text Box	Text1	Stores the filename.
Label	Label	Label of Text1.

11-12 The list of controls in the PRINTFL1.FRM form.

Application (Code) Name: PRINTFL1
Form # 1
Version: 1.0 Date: July 8, 1992

Original control name	Property	New setting
Command1	CtlName	PrintBtn
	Caption	Print
	Default	TRUE
Command2	CtlName	QuitBtn
	Caption	Quit
	Cancel	TRUE
Text1	CtlName	TextBox
	Text	(empty string)
Label1	CtlName	FilenameLbl
	Caption	Filename

11-13 The updated settings for the controls of form PRINTFL1.FRM.



11-14 A sample session with the text file printing program.

The PrintBtn_Click procedure Prints the text file specified in TextBox.Text. The procedure performs the following steps:

- Searches for a first inner space in the TextBox.Text property. If found, the Text property is truncated starting with that inner space.
- Obtains the filename, initializes the page number, and sets up the proper error-handling trap before opening the text file.
- Opens the text file. If the file is not found, the error-handling routine is triggered.

Listing 11-4 The code attached to the PRINTFL1.FRM form.

```
DECLARE SUB PrintHeading ()
'$INCLUDE: 'CONSTANT.BI'
CONST MAX_LINES = 55

DIM SHARED Filename AS STRING
DIM SHARED CurrentLineNumber AS INTEGER
DIM SHARED Page AS INTEGER


SUB PrintBtn_Click ()
    DIM TextLine AS STRING
    DIM I AS INTEGER
    ' exit sub if text box is empty
    IF TextBox.Text = "" THEN EXIT SUB
    ' find the first internal space
    I = INSTR(TextBox.Text, " ")
    ' truncate from the first space location and beyond
    IF I > 0 THEN
        TextBox.Text = LEFT$(TextBox.Text, I - 1)
    END IF
    ' get the filename from the text box
    Filename = TextBox.Text
    Page = 0 ' initialize the page number
    ON LOCAL ERROR GOTO BadFile' set error-handling trap
    OPEN Filename FOR INPUT AS #1 ' open file for input
    ON LOCAL ERROR GOTO 0' disable error-handling trap
    PrintHeading ' print the heading of the first page
    ' read lines from the text file
    DO WHILE NOT EOF(1)
        LINE INPUT #1, TextLine
        ' update line counter
        CurrentLineNumber = CurrentLineNumber + 1
        ' line counter exceed page size?
        IF CurrentLineNumber > MAX_LINES THEN
            Printer.NEWPAGE ' new page
            PrintHeading  ' print heading
        END IF
        Printer.PRINT TextLine ' print the current line
    LOOP
    Printer.NEWPAGE ' eject the last page
    Printer.ENDDOC' release print device
    CLOSE #1 ' close the file buffer
    BEEP
    EXIT SUB
    '****************** Error-Handling Statements **************
    BadFile:
        BEEP

        MSGBOX "Cannot open file " + Filename, 0, "File I/O Error"
```

Listing 11-4 Continued.

```
    ON LOCAL ERROR GOTO 0
    RESUME ExitSub
ExitSub:
END SUB

SUB PrintHeading ()
      CONST TAB_SIZE = 65
      Page = Page + 1
      Printer.PRINT Filename; TAB(TAB_SIZE);
      Printer.PRINT "Page "; FORMAT$(Page, "###")
      Printer.PRINT FORMAT$(NOW, "hh:mm");
      Printer.PRINT TAB(TAB_SIZE);
      Printer.PRINT FORMAT$(NOW, "MM-DD-YYYY")
      Printer.PRINT
      Printer.PRINT
      CurrentLineNumber = 4
END SUB

SUB QuitBtn_Click ()
    END
END SUB

SUB TextBox_KeyPress (KeyAscii AS INTEGER)
    KeyAscii = ASC(UCASE$(CHR$(KeyAscii)))
END SUB
```

- Uses a DO-WHILE loop to read the lines of the text file. The current number of lines is monitored for proper form feed operations. The Printer.NEWPAGE statement. The Printer.Print TextLine statement sends the current text line to the printer.
- Uses the Printer.ENDDOC statement to eject the last page.
- Closes the file buffer and exits the procedure.

The TextBox_KeyPress procedure Converts all the letters you type into uppercase characters.

12

Managing mouse events

The Click event is perhaps the most frequently used mouse-related event. However, it is by no means the only one. This chapter looks at the other mouse events and how you can use them in your Visual Basic applications.

The mouse events

Visual Basic offers the following mouse events that are related to the mouse status:

The MouseDown event Occurs when you press on any mouse button. A typical event-handling procedure for the MouseDown event is shown below:

```
SUB object_MouseDown(Button AS INTEGER, Shift AS INTEGER,
    X AS SINGLE, Y AS SINGLE)
```

The Button parameter is bit oriented. The three least-significant bits give the status of the mouse button. The other bits of the Integer-typed parameter are not used. Table 12-1 shows the arguments for the Button parameter. Visual Basic ignores the middle button.

The Shift parameter indicates whether or not the SHIFT, CTRL, and ALT keys were held down when the MouseDown event occurred. Table 12-2 shows the arguments for the Shift parameter.

The X and Y parameters report the coordinates of the mouse when the MouseDown event occurs.

The MouseUp event Occurs when you release on any mouse button. A typical event-handling procedure for the MouseUp event is shown below:

```
SUB object_MouseUp(Button AS INTEGER, Shift AS INTEGER,
    X AS SINGLE, Y AS SINGLE)
```

The parameters of the MouseUp events are identical to those of the MouseDown event.

The MouseMove event Occurs when you move the mouse. A typical event-handling procedure for the MouseMove event is shown below:

```
SUB object_MouseMove (Button AS INTEGER, Shift AS INTEGER,
    X AS SINGLE, Y AS SINGLE)
```

The parameters of the MouseMove events are identical to those of the MouseDown event.

The free-drawing program

Let's look at a program that allows you to draw lines with the mouse. The lines are made up of characters, usually extended ASCII characters. The program has the following features:

- The program's form has no controls drawn on it.
- The program uses menu items to support the following operations:
  - ~ Exiting from the program.
  - ~ Clearing the form.
  - ~ Printing the form.
  - ~ Selecting the drawing characters.
- You can draw characters by either clicking either mouse button or holding either button and moving the mouse.
- To support more drawing characters, the program offers different characters when you press the SHIFT and CTRL keys. When you press the ALT key the program switches to character erasing mode.
- The menu options for selecting the drawing (or fill) characters use input dialog boxes to display the current fill character and request a new one. To specify a new fill character you need to enter its ASCII code. Therefore, you should have an ASCII table handy.

Figures 12-1 and 12-2 show the specifications for the MOUSEV1.MAK project file and the menu of the MOUSEV1.FRM form. The program's form only contains a menu. Figure 12-3 shows a sample session with the MOUSEV1 program.

Listing 12-1 shows the code attached to the MOUSEV1.FRM form. The general declarations section contains the declarations of four variables. The FillChar, ShiftFillChar, and CtlFillChar variables store the drawing (or fill) characters when no shifted key is pressed, when the SHIFT key is pressed, and when the CTRL key is pressed, respectively. The MouseIsDown variable

Table 12-1 The arguments for the Button parameter.

Bit number	Value when on	Meaning
(none)	0	No button is pressed.
0	1	The left button generated the MouseDown event.
1	2	The right button generated the MouseDown event.
0 and 1	3	The right and left buttons generated the MouseDown event.

Table 12-2 The arguments for the Shift parameter.

Bit number	Value when on	Meaning
(none)	0	No key was pressed.
0	1	The SHIFT key is pressed.
1	2	The CTRL key is pressed.
2	4	The ALT key is pressed.
0 and 1	3	The SHIFT and CTRL keys are pressed.
0 and 2	5	The SHIFT and ALT keys are pressed.
1 and 2	6	The CTRL and ALT keys are pressed.
0, 1, and 2	7	All three keys are pressed.

Application name: Line-Drawing Demo
Application Code Name: MOUSEV1
Version: 1.0 Date Created: July 16, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	MOUSEV1.MAK
Form 1	MOUSEV1.FRM

12-1 The basic specifications for the MOUSEV1.MAK project file.

acts as a flag to detect whether or not a mouse button is pressed down.

The form includes the following event-handling procedures:

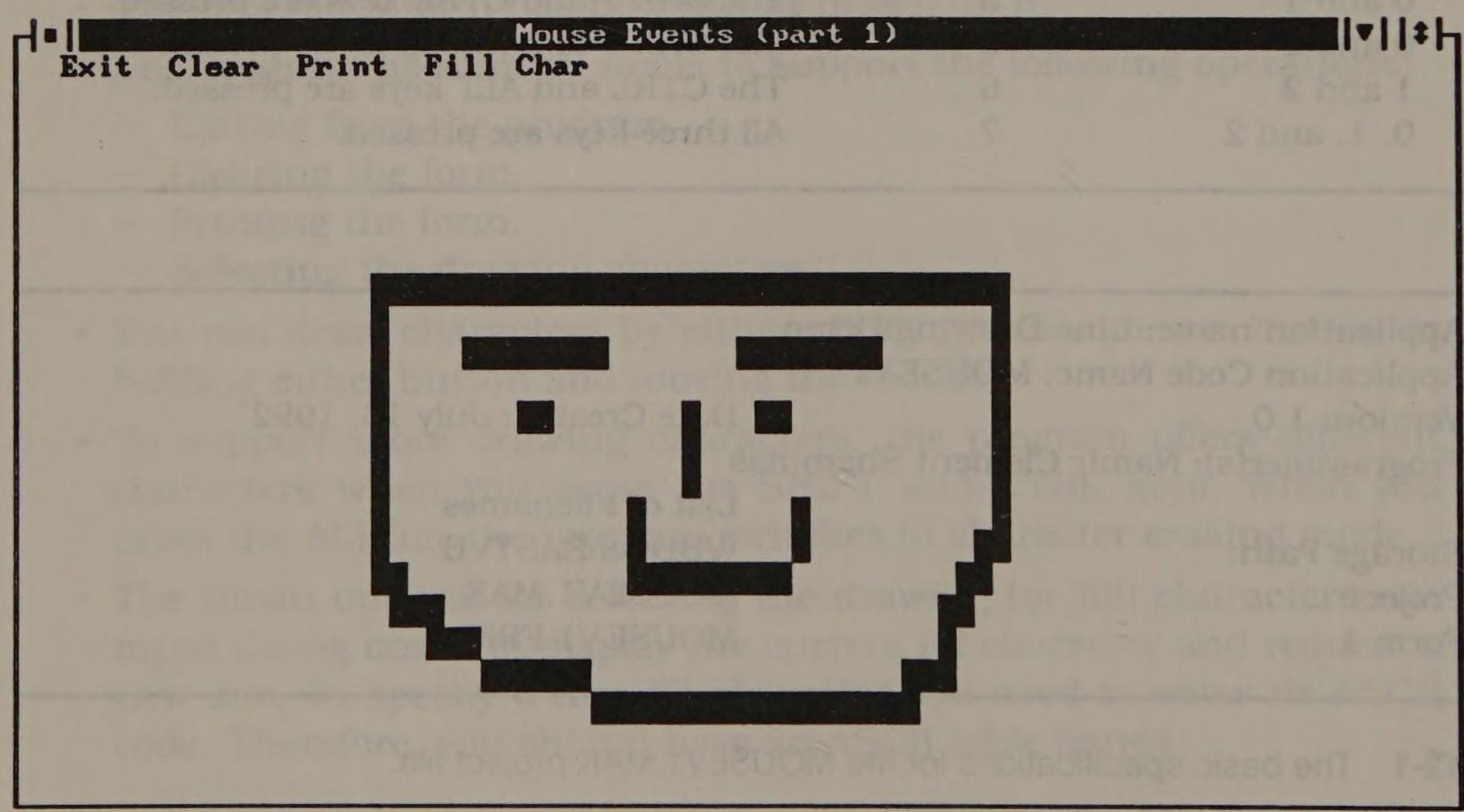
The ClearCom_Click procedure Clears the form by simply invoking the CLS method.

The NormalCom_Click procedure Prompts you for the ASCII code of the fill character when no shifted keys are pressed. The procedure calls the INPUTBOX$ function to display the prompt and the ASCII code for the current fill

Form #1 Form Filename: MOUSEV2.FRM
Version: 1.0 Date: July 16, 1992

Caption	Property	Setting	Purpose
&Exit	CtlName	ExitCom	Exits application
&Clear	CtlName	ClearCom	Clears the form
&Print	CtlName	PrintCom	Prints the form
&Fill Char	CtlName	FillCharCom	Select a fill character
Normal	CtlName	NormalCom	Select fill char for normal drawing
Shifted	Accelerator	CTRL+N	Select fill char for
	indented	once	drawing while holding
	CtlName	ShiftedCom	SHIFT key
Control	Accelerator	CTRL+S	Select fill char for
	indented	once	drawing while holding
	CtlName	ControlCom	CTRL key
	Accelerator	CTRL+C	
	indented	once	

12-2 List of control objects for MOUSEV1.FRM form.



12-3 A sample session with the MOUSEV1 program.

char. The procedure converts your input into an ASCII code, whose character is assigned to the FillChar variable. If you input an empty string, the procedure assigns the character whose ASCII code is 219 to the FillChar.

The ControlCom_Click procedure Similar to the NormalCom_Click procedure; the main difference is that this procedure manipulates the string stored in variable CtlFillChar, instead of FillChar.

The Form_Load procedure Performs a series of initialization. This task includes assigning the initial fill characters to variables FillChar, ShiftFillChar, and CtlFillChar. The procedure also assigns FALSE to the MouseIsDown, turns on the auto-redraw property of the form, maximizes the window, and selects the cross character as the mouse pointer.

The Form_MouseDown procedure Draws or erases a character at the current mouse location. The procedure first tests if variable MouseIsDown is FALSE. If this condition is true, the procedure draws or erases a character. The routine uses an extended IF-THEN-ELSE statement to examine the value of the Shift parameter and to determine which shifted keys are pressed. This examination results in drawing a character using the appropriate fill character, or simply erasing the character at the current cursor location. The cursor location is provided by the parameters X and Y. The procedure assigns X and Y to properties CurrentX and CurrentY, respectively, to move the character insertion point to the current cursor location.

The Form_MouseMove procedure Draws or erases characters while the mouse is moving. The code for this routine is similar to that of Form_MouseDown. The difference is that this routine does not alter the value of variable MouseIsDown.

The Form_MouseUp procedure Assigns FALSE to the variable MouseIsDown. This value prevents further movement of the mouse to continue to draw characters (before you press a mouse button).

The PrintCom_Click Prints the form by merely calling the PRINTFORM method.

The ShiftedCom_Click procedure Similar to the NormalCom_Click procedure; the main difference is that this procedure manipulates the string stored in variable ShiftFillChar, instead of FillChar.

Run the program and experiment with drawing various lines. Draw some lines without using the SHIFT, CTRL, or ALT keys. Then try and draw other lines while pressing the SHIFT key. Now draw more lines while holding the CTRL key. Press the ALT key and click the mouse button to erase a character. To erase multiple characters, hold down the mouse button, hold down the ALT, and move the mouse.

The mouse-pointer demo program

The programs presented so far use the default mouse cursor (the block). Visual Basic allows you to select other shapes for the mouse cursor. Table 12-3 shows the list of mouse pointers, their shapes, and ASCII values. To alter the mouse pointer use the MousePointer property, whose general syntax is

```
[form.] [object.]MousePointer [=setting%]
```

The value of setting% must be in the range of 0 to 12.

Listing 12-1 The code attached to the MOUSEV1.FRM form.

```
'$INCLUDE: 'CONSTANT.BI'
DIM SHARED FillChar AS STRING
DIM SHARED ShiftFillChar AS STRING
DIM SHARED CtlFillChar AS STRING
DIM SHARED MouseIsDown AS INTEGER

SUB ClearCom_Click ()
  CLS
END SUB

SUB ControlCom_Click ()
  DIM S AS STRING
  DIM i AS INTEGER
  S = "Enter the ASCII code for the Control filler character"
  S = INPUTBOX$(S, "Integer Input", STR$(ASC(CtlFillChar)))
  IF S = "" THEN S = "219"
  i = VAL(LTRIM$(S))
  IF i = 0 THEN i = 219
  CtlFillChar = CHR$(i)
END SUB

SUB ExitCom_Click ()
  END
END SUB

SUB FillCharCom_Click ()
  DIM S AS STRING
  DIM i AS INTEGER
  S = "Enter the ASCII code for the filler character"
  S = INPUTBOX$(S, "Integer Input", "219")
  IF S = "" THEN S = "219"
  i = VAL(S)
  IF i = 0 THEN i = 219
  FillChar = CHR$(i)
END SUB

SUB Form_Load ()
  ' initialize the fill characters
  FillChar = CHR$(219)
  ShiftFillChar = CHR$(178)
  CtlFillChar = CHR$(223)
  MouseIsDown = FALSE
  AutoReDraw = TRUE
  WindowState = 2 ' maximize the form
  MousePointer = 2 ' cross pointer
END SUB

SUB Form_MouseDown (Button AS INTEGER, Shift AS INTEGER, X AS SINGLE, Y AS
SINGLE)
  IF MouseIsDown = FALSE THEN
    CurrentX = X
    CurrentY = Y
    IF Shift AND 1 THEN
      PRINT ShiftFillChar
    ELSEIF Shift AND 2 THEN
      PRINT CtlFillChar
    ELSEIF Shift AND 4 THEN
      PRINT " "
    ELSE
      PRINT FillChar
    END IF
    MouseIsDown = TRUE
  END IF
END SUB

SUB Form_MouseMove (Button AS INTEGER, Shift AS INTEGER, X AS SINGLE, Y AS
SINGLE)
```

```
  IF MouseIsDown = TRUE THEN
    CurrentX = X
    CurrentY = Y
    IF Shift AND 1 THEN
      PRINT ShiftFillChar
    ELSEIF Shift AND 2 THEN
      PRINT CtlFillChar
    ELSEIF Shift AND 4 THEN
      PRINT " "
    ELSE
      PRINT FillChar
    END IF
  END IF
END SUB

SUB Form_MouseUp (Button AS INTEGER, Shift AS INTEGER, X AS SINGLE, Y AS
SINGLE)
  MouseIsDown = FALSE
END SUB

SUB NormalCom_Click ()
  DIM S AS STRING
  DIM i AS INTEGER
  S = "Enter the ASCII code for the filler character"
  S = INPUTBOX$(S, "Integer Input", STR$(ASC(FillChar)))
  IF S = "" THEN S = "219"
  i = VAL(LTRIM$(S))
  IF i = 0 THEN i = 219
  FillChar = CHR$(i)
END SUB

SUB PrintCom_Click ()
  PRINTFORM
END SUB

SUB ShiftedCom_Click ()
  DIM S AS STRING
  DIM i AS INTEGER
  S = "Enter the ASCII code for the Shift filler character"
  S = INPUTBOX$(S, "Integer Input", STR$(ASC(ShiftFillChar)))
  IF S = "" THEN S = "219"
  i = VAL(LTRIM$(S))
  IF i = 0 THEN i = 219
  ShiftFillChar = CHR$(i)
END SUB
```

Let me present a simple program that manipulates and changes the shape of the mouse pointer. Figures 12-4 through 12-6 show the specifications for the MOUSEV2.MAK project file and the menu of the MOUSEV2.FRM form. The program's form only contains a menu. Figure 12-7 shows a sample session with the MOUSEV2 program.

The program demonstrates how the mouse pointer changes when you move it inside a control and when you click. If you look at FIG. 12-6 you see the custom values for the MousePointer properties for the text box, the command button, and the picture box. When you move the mouse pointer inside these controls it changes shape according to the value of the MousePointer property for each control. In addition, the program changes the shape of the mouse pointer by clicking on the mouse buttons. When you click on the left mouse button the MousePointer property is incremented by 1. By contrast, when you click the right mouse button, the MousePointer property is decre-

Table 12-3 The various mouse pointers.

Mouse pointer value	Shape	ASCII value
0	default	
1	Block (■)	219
2	Cross (+)	197
3	IBeam (I)	073
4	(☺)	002
5	Size (*)	015
6	Left arrow (←)	027
7	Size north/south (↕)	018
8	Right arrow (→)	026
9	Size east/west (↔)	029
10	Up arrow (↑)	024
11	Hourglass (X)	088
12	Down arrow (↓)	025

Application Name: The Mouse-Pointer Demo
Application Code Name: MOUSEV2
Version: 1.0 Date Created: July 16, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	MOUSEV2.MAK
Form 1	MOUSEV2.FRM

12-4 The general specifications for the MOUSEV2.MAK project.

Form # 1 Form Filename: MOUSEV2.FRM
Version: 1.0 Date: July 10, 1992

Control Object Type	Default CtlName	Purpose
Text Box	Text1	A text box
Command Button	Command1	The Quit command button
Picture	Picture1	A picture box
Label	Label1	The label for the picture box
	Label2	The label for the text box

12-5 List of control objects for the MOUSEV2.FRM form.

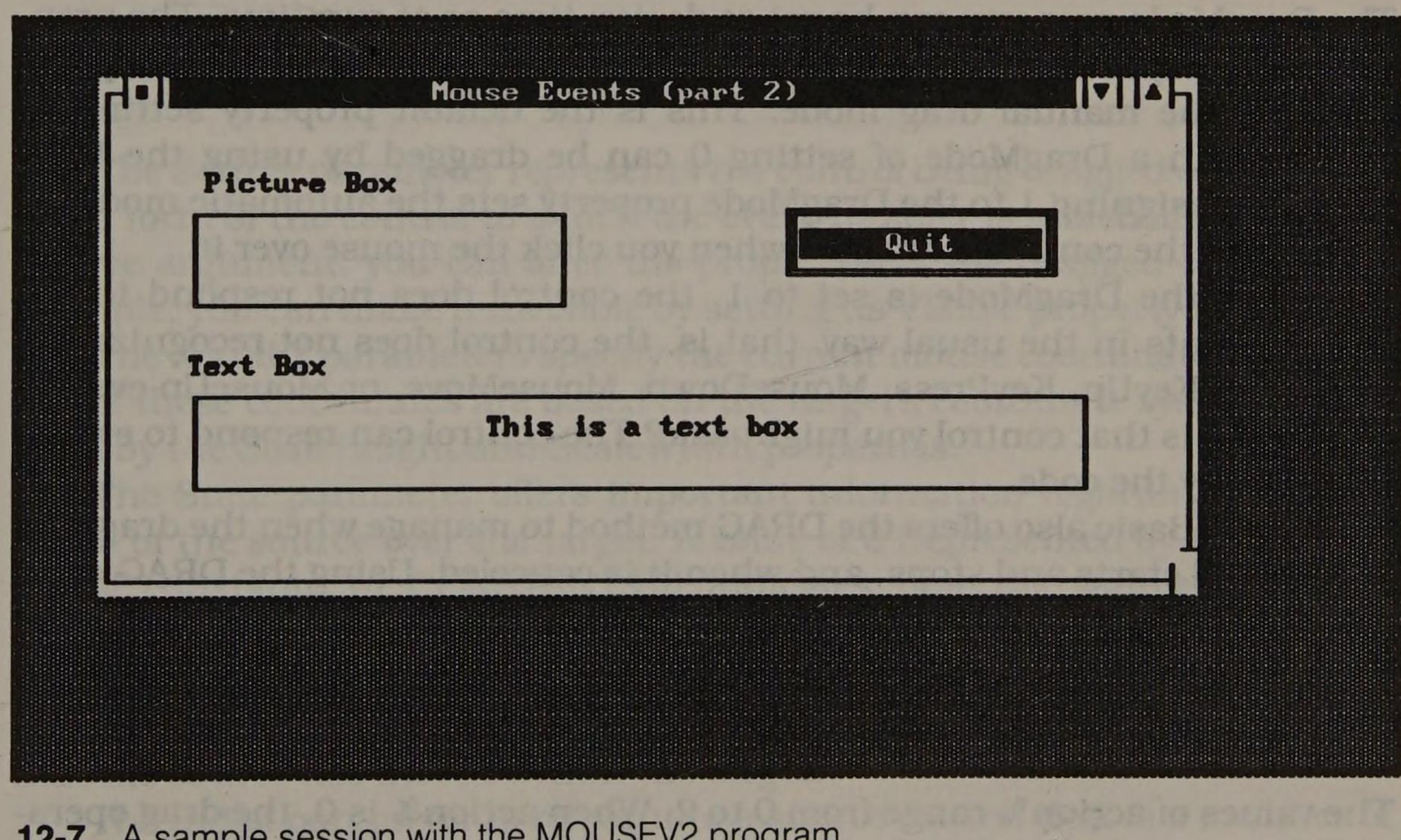
mented by 1. If you click quickly, you end up generating double-click events that do not change the shape of the mouse pointer.

Listing 12-2 contains the code attached to the MOUSEV2.FRM form. The short listing has the following relevant procedures:

Application (Code) Name: MOUSEV2
Form # 1
Version: 1.0 Date: July 10, 1992

Original Control Name	Property	New Setting
Form	Caption	Mouse Events (part 2)
Text1	CtlName	TextBox
	Text	This is a text box
	MousePointer	11 - Hourglass
Command1	CtlName	TextBox
	Caption	Quit
	MousePointer	9 - Size east/west
Picture1	MousePointer	2 - Cross
Label1	CtlName	PictureBoxLbl
	Caption	Picture Box
Label2	CtlName	TextBoxLbl
	Caption	Text Box

12-6 The updated settings of MOUSEV2.FRM form.



12-7 A sample session with the MOUSEV2 program.

The form_Load procedure Performs simple form initialization which assigns 1 to the variable PointerIndex and the form's MousePointer property.

The Form_MouseDown procedure Alters the shape of the mouse cursor. The procedure examines the value of the Button parameter to determine which button you clicked. If the value of Button is 1 then you clicked the left button. Consequently, the value of the PointerIndex is increased by 1. The MOD

operator allows the increase in variable PointerIndex to be in the range of 0 to 12. If the value of Button is not 1 then you clicked the right button. Consequently, the value of the PointerIndex is decreased by 1. If the value of PointerIndex becomes negative, the procedure assigns 12 to the variable.

Dragging and dropping controls

One of the features of graphical user interface (GUI) environments is the ability to drag controls. In Visual Basic you can have DOS applications that drag a control over on a form or on other controls. The dragging ends when you release the mouse button and effectively drop the control on another control or on the form. Visual Basic calls the control being dragged the *Source*. In addition, the control or the form over which a source control is dropped is called the *Target*.

Visual Basic offers the DragMode property to set the manual or automatic mode for a drag-and-drop operation. This property is available for a wide variety of controls. The general syntax for the DragMode property is:

```
[form.]control.DragMode [=mode%]
```

The DragMode property can be set at design time or at run time. The property can accept one of two values, 0 or 1. Assigning 0 to the DragMode property sets the manual drag mode. This is the default property setting. A control with a DragMode of setting 0 can be dragged by using the Drag method. Assigning 1 to the DragMode property sets the automatic mode. In this mode, the control is dragged when you click the mouse over it.

When the DragMode is set to 1, the control does not respond to the mouse events in the usual way, that is, the control does not recognize the KeyDown, KeyUp, KeyPress, MouseDown, MouseMove, or MouseUp events. What good is that control you might ask? The control can respond to events triggered by the code.

Visual Basic also offers the DRAG method to manage when the dragging of a control starts and stops, and when it is canceled. Using the DRAG command offers better control than manipulating the DragMode property. The general syntax for the Drag method is:

```
[control.]DRAG [action%]
```

The values of action% range from 0 to 2. When action% is 0, the drag operation is canceled. When action% is 1, dragging a control begins. Dragging stops when the action% is 2 and the control is dropped (causing a DragDrop event to occur).

Visual Basic offers the following two events that are related to the drag-and-drop operations:

The DragOver event Occurs when the control is being dragged over a target (either a form or another control). The general syntax for the DragOver event-handling procedure has two forms:

Listing 12-2 The code attached to the MOUSEV2.FRM form.

```
DIM SHARED PointerIndex AS INTEGER

SUB Form_Load ()
  ' initialize the mouse pointer
  PointerIndex = 1
  MousePointer = PointerIndex
END SUB

SUB Form_MouseDown (Button AS INTEGER, Shift AS INTEGER, X AS SINGLE, Y AS
SINGLE)
  IF Button = 1 THEN
    PointerIndex = PointerIndex MOD 12 + 1
  ELSE
    PointerIndex = PointerIndex - 1
    IF PointerIndex = 0 THEN PointerIndex = 12
  END IF
  MousePointer = PointerIndex
END SUB

SUB QuitBtn_Click ()
  END
END SUB
```

SUB Form_DragOver(**Source** AS CONTROL, **X** AS SINGLE, **Y** AS SINGLE, **State** AS INTEGER)

SUB **control**_DragOver([**Index** AS INTEGER,] **Source** AS CONTROL, **X** AS SINGLE, **Y** AS SINGLE, **State** AS INTEGER)

The Source parameter represents the control being dragged over the target (a form of the control to which the event-handler is attached). Using the Source argument, you can alter the properties of the dragged control. For example, you can make it invisible by setting its Visible property to FALSE.

The X and Y parameters specify the current mouse coordinates. The values of these coordinates are based on the target's coordinate system, as defined by the ScaleHeight and ScaleWidth properties.

The State parameter offers important information regarding the transition of the source over the target. A value of 0 (represented by the ENTER global constants in CONSTANT.BI) indicates that the source control is dragged within range of the target. A value of 1 (represented by the LEAVE global constant) indicates that the source control is dragged out of range of the target. A value of 2 (represented by the OVER global constant) signals that the source has moved within the target.

The Index parameter is relevant to a source control that is a member of a control array; the Index parameter enables you to select the proper source control.

The DragDrop event Occurs when the source control is dropped over a target by releasing the mouse button, or by a Drag 2 statement. The general syntax for the DragDrop event-handling procedure has two forms:

SUB Form_DragDrop(**Source** AS CONTROL, **X** AS SINGLE, **Y** AS SINGLE, **State** AS INTEGER)

SUB control_DragDrop([Index AS INTEGER,] Source AS CONTROL, X AS SINGLE, Y AS SINGLE, State AS INTEGER)

The Source parameter represents the control being dragged over the target (a form or the control to which the event-handler is attached).

The X and Y parameters specify the current mouse coordinates. The values of these coordinates are based on the target's coordinate system, as defined by the ScaleHeight and ScaleWidth properties.

The Index parameter is relevant to a source control that is a member of a control array. The Index parameter enables you to select the proper source control.

☞ The source controls are moved using the Move method. These methods are usually placed in a DragDrop event-handler (typically, the Form_DragDrop procedure). Otherwise, the source controls remain where they were when you started dragging them.

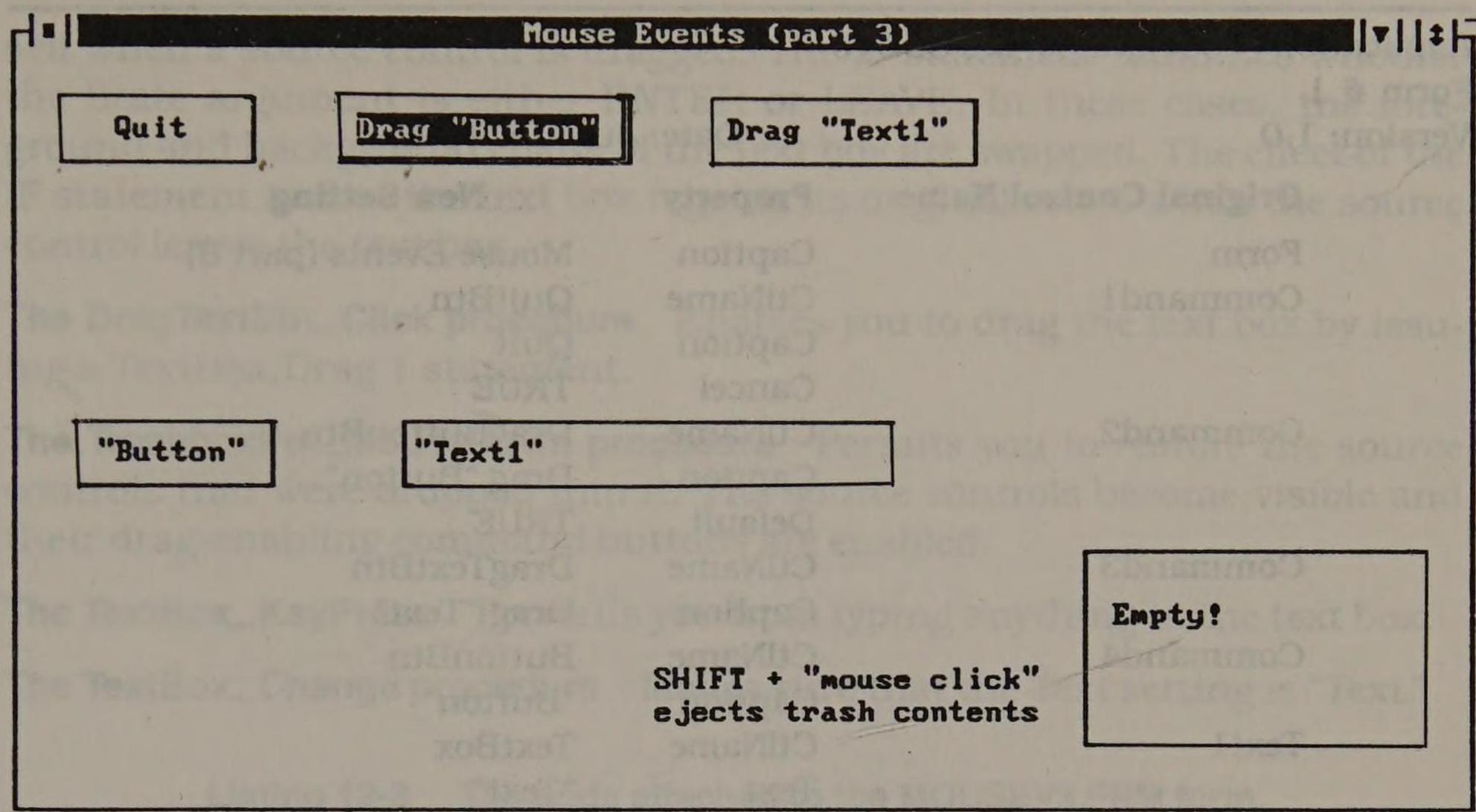
Let's look at a program that illustrates a simple dragging of controls. The next program, MOUSEV3, illustrates how to drag a command button and a text button and then drop them in a trash-box picture box. The program, shown in FIG. 12-8, contains three controlling command buttons. The button labeled Quit exits the application. The button labeled Drag "Button" triggers the dragging of a button marked "Button." The button labeled Drag "Text" triggers the dragging of the text box containing "Text." At the lower-right corner is a picture box that acts as a trash can (admittedly, you have to use your imagination).

Initially, the trash can is empty. Once you drop either the "Button" command button or the text box, these controls disappear and the text in the picture box changes from Empty! into the Have Trash! message. If you click the mouse on the picture box while pressing the SHIFT key, the invisible controls (presumed in the trash can) reappear and the trash can changes text back to the Empty! message. When you drop a source control in the trash box, its drag-enabling command button is disabled. When you empty the trash, these drag-enabling command buttons are enabled.

The "Button" command button and the text box can only be moved after you click on their respective drag-enabling command buttons. You can drop the source controls on the form. Once you drop these controls on the form, they cannot be dragged without clicking again on the drag-enabling buttons. I programmed the "Button" command button to respond with a message box when clicked on. I also programmed the text box to maintain the "Text" (with the quotes). When not being dragged these controls respond to the mouse and keyboard events as programmed.

Figures 12-9 through 12-11 contain the specifications for the MOUSEV3.MAK project file and the MOUSEV3.FRM form file.

Listing 12-3 shows the code attached to the MOUSEV3.FRM form. The relevant procedures are:



12-8 The MOUSEV3.FRM at design time.

Application Name: Control Dragging Demo
Application Code Name: MOUSEV3
Version: 1.0 Date Created: July 16, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	MOUSEV3.MAK
Form 1	MOUSEV3.FRM

12-9 The general specifications for the MOUSEV3.MAK project.

Form # 1 Form Filename: MOUSEV3.FRM
Version: 1.0 Date: July 16, 1992

Control Object Type	Default CtlName	Purpose
Command Button	Command1	Exits the demo program
	Command2	Triggers dragging Command4
	Command3	Triggers dragging Text1
	Command4	The dragged command button
Text box	Text1	The dragged text box
Picture Box	Picture1	The trash box
Label	Label1	Trash box label that offer help on ejecting the trash

---

12-10 The list of control objects for the MOUSEV3.FRM form.

Application (Code) Name: MOUSEV3
Form # 1
Version: 1.0 Date: July 16, 1992

Original Control Name	Property	New Setting
Form	Caption	Mouse Events (part 3)
Command1	CtlName	QuitBtn
	Caption	Quit
	Cancel	TRUE
Command2	CtlName	DragButtonBtn
	Caption	Drag "Button"
	Default	TRUE
Command3	CtlName	DragTextBtn
	Caption	Drag "Text"
Command4	CtlName	ButtonBtn
	Caption	"Button"
Text1	CtlName	TextBox
	Text	"Text"
Label1	CtlName	TrashLbl
	Caption	SHIFT + "mouse click" to eject trash
Picture1	CtlName	TrashboxPct

12-11 The update setting for the MOUSEV3.FRM form.

The Form_Load procedure Sets the window state and writes Empty! in the picture box.

The TrashboxPct_DragDrop procedure Performs the following tasks:

- Hides the Source control by setting its Visible property to False.
- Disables the associated drag-enabling command button. An IF-THEN-ELSEIF statement uses the TYPEOF-IS operator to detect the control type of the Source argument.
- Writes Have Trash! in the trash can picture box.

☞ **The Form_DragDrop procedure** Responsible for actually moving the source control. The Source.Move method enables this procedure to relocate either the command button or the text box. Without the Move method, the source controls remain in their original places after they are dropped.

The DragButtonBtn_Click procedure Enables you to drag the "Button" command button by using the ButtonBtn.Drag method with an argument of 1.

The ButtonBtn_Click procedure Displays a message box when you click on the "Button" command button.

The TextBox_DragOver procedure Specifies what happens to the text box it-

self when a source control is dragged. The IF statement examines whether the State argument is either ENTER or LEAVE. In these cases, the foreground and background colors of the text box are swapped. The effect of the IF statement is that the text box regains its original colors when the source control leaves the text box.

The DragTextBtn_Click procedure Enables you to drag the text box by issuing a TextBox.Drag 1 statement.

The TrashboxPct_MouseDown procedure Permits you to restore the source controls that were dropped into it. The source controls become visible and their drag-enabling command buttons are enabled.

The TextBox_KeyPress Prevents you from typing anything in the text box.

The TextBox_Change procedure Makes sure that the Text setting is "Text."

Listing 12-3 The code attached to the MOUSEV3.FRM form.

```
'$INCLUDE: 'CONSTANT.BI'

SUB ButtonBtn_Click ()
    MSGBOX "I am a dud!", 0, "For Your Precious Information"
END SUB

SUB DragButtonBtn_Click ()
    ButtonBtn.DRAG 1
END SUB

SUB DragTextBtn_Click ()
    TextBox.DRAG 1
END SUB

SUB Form_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)
      Source.MOVE (X - Source.Width / 2), (Y - Source.Height / 2)
END SUB

SUB Form_Load ()
      WindowState = 2
      TrashboxPct.AutoReDraw = TRUE
      TrashboxPct.CurrentX = 1
      TrashboxPct.CurrentY = 1
      TrashboxPct.PRINT "Empty!"
END SUB

SUB QuitButton_Click ()
    END
END SUB

SUB TextBox_Change ()
    TextBox.Text = CHR$(34) + "Text" + CHR$(34)
END SUB

SUB TextBox_DragOver (Source AS CONTROL, X AS SINGLE, Y AS SINGLE, State AS
INTEGER)
    DIM TColor AS LONG
    IF (State = ENTER) OR (State = LEAVE) THEN
        TColor = TextBox.BackColor
        TextBox.BackColor = TextBox.ForeColor
        TextBox.ForeColor = TColor
```

Listing 12-3 Continued.

```
    END IF
END SUB

SUB TextBox_KeyPress (KeyAscii AS INTEGER)
    KeyAscii = 0
END SUB

SUB TrashboxPct_Click ()

END SUB

SUB TrashboxPct_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)
    Source.Visible = FALSE
    IF TYPEOF Source IS TextBox THEN
        DragTextBtn.Enabled = FALSE
    ELSEIF TYPEOF Source IS CommandButton THEN
        DragButtonBtn.Enabled = FALSE
    END IF
    TrashboxPct.CLS
    TrashboxPct.CurrentX = 1
    TrashboxPct.CurrentY = 1
    TrashboxPct.PRINT "Have Trash!"
END SUB

SUB TrashboxPct_MouseDown (Button AS INTEGER, Shift AS INTEGER, X AS SINGLE,
Y AS SINGLE)
    IF Shift AND 1 THEN
          TrashboxPct.CLS
          TranshboxPct.CurrentX = 1
          TranshboxPct.CurrentY = 1
          TrashboxPct.PRINT "Empty!"
          TextBox.Visible = TRUE
          ButtonBtn.Visible = TRUE
          DragTextBtn.Enabled = TRUE
          DragButtonBtn.Enabled = TRUE
    END IF
END SUB
```

Run the MOUSEV3 program and drag the "Button" command button and the text box around. Drop them in the trash can picture box and notice the change in text inside the picture box. Click on the picture box while pressing the SHIFT key and watch the source controls that you dropped earlier reappear. In fact, they appear in the last location before they were dropped in the picture box, because the picture box never moved them—it only made them invisible.

13

Interactive Visual Basic programs

This chapter looks at writing Visual Basic programs that are more interactive, including the ability of Visual Basic programs to exchange data with the Visual Basic CLIPBOARD object. Another aspect of creating interactive Visual Basic programs deals with using timer controls to periodically trigger timer events. In addition, this chapter looks at how you can detect any key on your keyboard, including functions keys and cursor control keys.

Using the CLIPBOARD

Chapter 7 discussed the features of text boxes. Part of the discussion focused on working with selected text using the SelText, SelStart, and SelLength properties. The SelText property can be used to transfer data between a Visual Basic program and the CLIPBOARD, a special object. The CLIPBOARD is a kind of data storage area where information is sent and later retrieved. The CLIPBOARD serves as the holding place where text is stored when you perform a copy or cut operation. Pasting is an operation that retrieves the contents of the CLIPBOARD into the current text box control.

The Visual Basic CLIPBOARD object is a special object that is void of properties and events—it has only a few methods for manipulating its contents. The Visual Basic CLIPBOARD uses the following methods:

The SETTEXT method Copies text onto the CLIPBOARD, overwriting the previous contents of the CLIPBOARD. The general syntax for the SETTEXT method is:

```
CLIPBOARD.SETTEXT stringExpression$
```

where stringExpression$ is the new content of the CLIPBOARD. Text can be copied onto the CLIPBOARD from any string or selected text using the SETTEXT method. Several examples are shown below:

```
CLIPBOARD.SETTEXT "Hello There!" ' string constant
CLIPBOARD.SETTEXT aTextLine$ ' string variable
CLIPBOARD.SETTEXT TextBox.Text ' a Text property
CLIPBOARD.SETTEXT TextBox.SelText ' copy selected text
```

The GETTEXT method A function that retrieves the text stored in the CLIPBOARD. The general syntax for the GETTEXT method is:

```
targetString$ = CLIPBOARD.GETTEXT()
```

Here are some examples of using the GETTEXT method:

```
aTextLine$ = CLIPBOARD.GETTEXT() ' to a string variable
TextBox.Text = CLIPBOARD.GETTEXT() ' to a Text property
TextBox.SelText = CLIPBOARD.GETTEXT() ' copy to selected text
```

Using the SETTEXT and GETTEXT methods you can write event-handling procedures that work with Copy, Cut, and Paste menu options that you might include in your Visual Basic applications. Typical code that transfers the text between the text box TextBox and the CLIPBOARD is shown below:

```
SUB CopyCom_Click ()
    CLIPBOARD.SETTEXT TextBox.SelText
END SUB

SUB CutCom_Click ()
    CLIPBOARD.SETTEXT TextBox.SelText
    TextBox.Text = ""
END SUB

SUB PasteCom_Click ()
    TextBox.SelText = CLIPBOARD.GETTEXT()
END SUB
```

The above sample code shows the interaction between the CLIPBOARD and a single text box. The SelText property plays a pivotal role in indicating the text that is cut, copied, or pasted.

☞ Visual Basic allows you to make better use of the CLIPBOARD object by allowing it to interact with any text box, as long as that text box has the focus. The ActiveControl property of the SCREEN object enables you to access the focused text box. The following set of procedures shows how a more general set of procedures is written to support the copy, cut, and paste menu options:

```
SUB CopyCom_Click ()
    CLIPBOARD.SETTEXT SCREEN.ActiveControl.SelText
END SUB

SUB CutCom_Click ()
```

```
    CLIPBOARD.SETTEXT SCREEN.ActiveControl.SelText
    SCREEN.ActiveControl.Text = ""
END SUB

SUB PasteCom_Click ()
    SCREEN.ActiveControl.SelText = CLIPBOARD.GETTEXT()
END SUB
```

The CLEAR method Clears the contents of the CLIPBOARD. The general syntax for using the CLEAR method is:

```
CLIPBOARD.CLEAR
```

Let's look at a program that uses the GETTEXT and SETTEXT methods to manipulate the clipboard. The WINEDIT1 program implements a simple text editor. This menu-driven program is derived from the FVIEW2C.MAK project and includes the ability to write text to a file.

The program features that are related to the CLIPBOARD include the following menu selections:

The Edit menu selection Contains the Cut, Copy, and Paste options. These options enable you to cut and copy text from the currently selected text box into the CLIPBOARD. The Paste option permits you to paste text from the CLIPBOARD into the currently selected text box.

The CLIPBOARD menu selection This menu contains the options to view and clear the CLIPBOARD, as well as to append the selected text to the CLIPBOARD. When you view the CLIPBOARD you actually copy its contents into the edit text box—the original text lines of the edit text box are stored in a variable. Placing the contents of the CLIPBOARD into the edit text box allows you to edit the clipboard. While you are viewing the CLIPBOARD the Clear and Append menu options are disabled, and the View CLIPBOARD caption is replaced with the caption View File. In addition, the filename text box shows the name CLIPBOARD, instead of the name of the edited file.

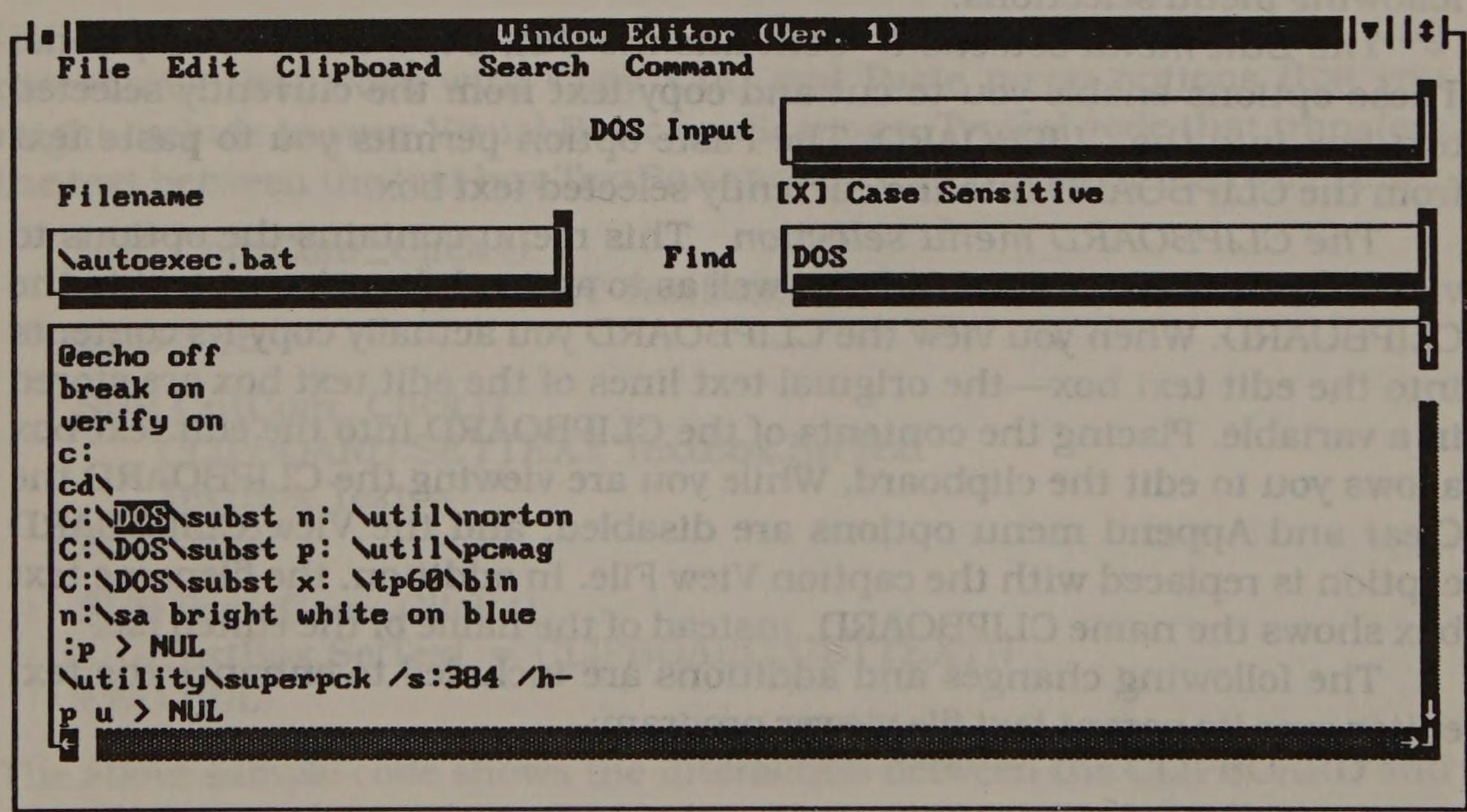
The following changes and additions are included to enhance the text editor over its parent text file viewer program:

- The Read option is renamed Load.
- The Save option is added. The name of the file in the Filename text box is used to save the edited text lines. By changing the filename in the text box you save the text lines into a new file; there is no need for a Save As menu option.
- The Read-Only option is added to the File menu. This option prevents you from writing the edited text lines to a file. Initially, this option is triggered by the Load_Form procedure to prevent you from accidentally overwriting a text file with a null string.
- The Add File and Delete File options have the CTRL-A and CTRL-D accelerator keys.
- The Search menu selection has a new option, namely, the Find Se-

lected Text. This option automatically copies the selected text of the edit text box into the Find text box, and then initiates the text search. The accelerator key for this new search option is the CTRL+S keys.

- The Case Sensitive check box is a new control that enables you to specify whether or not to use case-sensitive text search.

I suggest that you run the WINEDIT1 program first to get a feel for its features. I also recommend that you make a few copies of text files and experiment with editing these files (the WINEDIT1 program does not automatically create backup files). Use the options of the Edit and CLIPBOARD menu selections to manipulate blocks of text. A sample session with the WINEDIT1 program is shown in FIG. 13-1. Figures 13-2 to 13-5 contain the specifications of the WINEDIT1.MAK project file, the WINEDIT1.FRM form, and the menu structure.



13-1 A sample session with the WINEDIT1 program.

Listing 13-1 shows the code attached to the WINEDIT1.FRM form. The procedures relevant to the features that program WINEDIT1 added to FVIEW2C program are discussed here:

The FindCom_Click procedure Uses the following IF-THEN-ELSE to perform both case-sensitive and -insensitive text searches:

```
IF CaseSenseChk.Value = 1 THEN
    CurIndex = INSTR(CurIndex + 1, TextBox.Text, Find)
ELSE
    CurIndex = INSTR(CurIndex + 1, UCASE$(TextBox.Text),
```

Application Name: Menu-Driven Text File Editor
Application Code Name: WINEDIT1
Version: 1.0 Date Created: July 17, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	C:\VBDOS\EASYVB
Project	WINEDIT1.MAK
Form 1	WINEDIT1.FRM

13-2 General project specifications for the WINEDIT1 project.

Form # 1 Form Filename: WINEDIT1.FRM
Version: 1.0 Date: July 17, 1992

Control object type	Default CtlName	Purpose
Text Box	Text1	Contains the filename to read.
	Text2	Contains the text search string.
	Text3	Contains the DOS command line.
	Text4	Displays the text lines of a file.
Label	Label1	Label of Text1.
	Label2	Label of Text2.
	Label3	Label of Text3.
Check Box	Check1	Specifies case-sensitive search.

13-3 List of control objects for the WINEDIT1 project.

```
UCASE$(Find))
    END IF
```

The tested condition compares the CaseSenseChk.Value property with 1. If this condition is true, the case-sensitive search is conducted by using the TextBox.Text and Find strings. Otherwise, the UCASE$ function is applied to both these strings in order to carry out a case-insensitive search.

The FindSelTextCom_Click Copies the selected text of the edit text box and initiates a text search. If the selected text in the edit text box is not an empty string the Text property of FindBox is assigned the selected text. The procedure then calls the FindCom_Click procedure to conduct the text search.

The Form_Load procedure Initializes the application. This includes setting the window state to maximum and invoking the ReadOnlyCom_Click procedure to set the editor into read-only mode.

The SaveCom_Click procedure Saves the text lines of the edit text box into the specified filename. The procedure exits if the edit text box or the filename are empty strings. The statements in the procedure open the specified

Application (Code) Name: WINEDIT1
Form # 1
Version: 1.0 Date: July 17, 1992

Original control name	Property	New setting
Form	Caption	File Viewer (ver. 2A)
Text1	CtlName	FilenameBox
	Text	(empty string)
Text2	CtlName	FindBox
	Text	(empty string)
Text3	CtlName	DosBox
	Text	(empty string)
Text4	CtlName	TextBox
	Text	(empty string)
	MultiLine	TRUE
	ScrollBars	3 - both
Label1	CtlName	FilenameLbl
	Caption	Filename Input
Label2	CtlName	FindLbl
	Caption	Find Input
Label3	CtlName	DosLbl
	Caption	DOS Input
Check1	CtlName	CaseSenseChk
	Caption	Case Sensitive

13-4 The modified control settings for the WINEDIT1 project.

Form # 1 Form Filename: WINDEDIT1.FRM
Version: 1.0 Date: July 17, 1992

Caption	Property	Setting	Purpose
&File	CtlName	FileCom	
Load	CtlName	LoadCom	Reads a file in Text1.
	Accelerator	F3	
	indented	once	
Save	CtlName	SaveCom	Saves the text lines.
	Accelerator	F2	
	indented	once	
Toggle Read	CtlName	ToggleReadCom	Toggles Read menu option.
	indented	once	
Read-Only	CtlName	ReadOnlyCom	Sets file to read-only.
	indented	once	
Find File	CtlName	FindFileCom	Finds a file.
	Accelerator	CTRL-F	
	indented	once	
—	CtlName	Sep1	
	indented	once	
Add File	CtlName	AddFileCom	Adds a filename.
	Accelerator	CTRL-A	

Caption	Property	Setting	Purpose
	indented	once	
Delete File	CtlName	DeleteFileCom	Deletes a filename.
	Accelerator	CTRL-D	
	indented	once	
—	CtlName	Sep2	
	indented	once	
(noname)	CtlName	QuickFilename	
	Index	0	
	Visible	off	
	indented	once	
—	CtlName	Sep3	
	indented	once	
Quit	CtlName	QuitCom	Exits the application.
	Accelerator	CTRL-Q	
	indented	once	
&Edit	CtlName	EditCom	Copy, cut, and paste menu.
Cut	CtlName	CutCom	Cuts the selected text.
	Accelerator	CTRL-X	
	indented	once	
Copy	CtlName	CopyCom	Copies the selected text into the CLIPBOARD.
	Accelerator	CTRL-C	
	indented	once	
Paste	CtlName	PasteCom	Pastes the text from the CLIPBOARD.
	Accelerator	CTRL-V	
	indented	once	
&Clipboard	CtlName	CLIPBOARDCom	CLIPBOARD menu.
View CLIPBOARD	CtlName	ViewCom	Views the CLIPBOARD.
	indented	once	
Clear	CtlName	ClearCom	Clears the CLIPBOARD.
Append	CtlName	AppendCom	Appends selected text to the CLIPBOARD.
	indented	once	
&Search Find	CtlName	SearchCom	
		once	
	CtlName	FindCom	Finds text in Text1.
	Accelerator	CTRL-once I	
Find Selected	CtlName	FindSelTextCom	Finds selected text.
	Accelerator	CTRL-S	
Text	indented	once	
&Command	CtlName	CommandCom	
DOS Command	CtlName	DosCom	Executes a DOS command.
	Accelerator	CTRL-O	
	indented	once	
Toggle OS	CtlName	ToggleOsCom	Shows/hides command options.
	indented	once	

13-5 The menu structure for the WINDEDIT1 project.

file for output, and perform the file I/O error-trapping if you specify an invalid filename.

The CutCom_Click procedure Deletes the selected text in the text box that has the focus. This feature is implemented using the Screen.ActiveControl, mentioned earlier.

The CopyCom_Click procedure Copies a nonempty selected text from the focused text box into the CLIPBOARD.

The AppendCom_Click procedure Appends the selected text of the edit text box to the CLIPBOARD. A pair of new line and carriage return characters are inserted before the appended text.

The PasteCom_Click procedure Pastes text from the CLIPBOARD to the currently selected text box.

The Cut, Copy, and Paste menu options enable you to exchange text among the various text boxes of the program.

The ClearCom_Click Clears the CLIPBOARD using the CLEAR method.

The ViewCom_Click procedure Enables you to view and edit the contents of the CLIPBOARD. The edited text lines and the filename are stored in static local variables. The procedure toggles between viewing the text lines and the CLIPBOARD. The CLIPBOARD menu options are also affected. The procedure disables the Clear and Append options when the CLIPBOARD is being viewed. In addition, the ViewCom.Caption property is set to View CLIPBOARD.

When the edited text lines are restored, the CLIPBOARD menu options are also returned to their original condition.

The ReadOnlyCom_Click procedure Disables the Save menu options and places a check mark to the left of the Read-Only menu option.

Listing 13-1 The code attached to the WINEDIT1.FRM form.

```
DECLARE SUB FindCom_Click ()
DECLARE SUB LoadCom_Click ()
DECLARE SUB ReadOnlyCom_Click ()
'$INCLUDE: 'CONSTANT.BI'
DIM SHARED NumFiles AS INTEGER

SUB AddFileCom_Click ()
    DIM theFilename AS STRING
    theFilename = INPUTBOX$("Enter full file name: ")
    NumFiles = NumFiles + 1 ' increment the menu count
    LOAD QuickFilename(NumFiles) ' load a new menu command
    QuickFilename(NumFiles).Caption = theFilename
    QuickFilename(NumFiles).Visible = TRUE
END SUB

SUB AppendCom_Click ()
    DIM S AS STRING
    IF TextBox.SelText = "" THEN EXIT SUB
    S = Clipboard.GETTEXT()
    S = S + CHR$(13) + CHR$(10) + TextBox.SelText
```

```
    Clipboard.SETTEXT S
END SUB

SUB ClearCom_Click ()
    Clipboard.CLEAR
END SUB

SUB CopyCom_Click ()
    IF Screen.ActiveControl.SelText <> "" THEN
        Clipboard.SETTEXT Screen.ActiveControl.SelText
    END IF
END SUB

SUB CutCom_Click ()
    Clipboard.SETTEXT Screen.ActiveControl.SelText
    Screen.ActiveControl.SelText = ""
END SUB

SUB DeleteFileCom_Click ()
    DIM N AS INTEGER: DIM I  AS INTEGER
    N = VAL(INPUTBOX$("Enter number to delete: "))
    IF (N > 0) AND (N <= NumFiles) THEN
        FOR I = N TO NumFiles - 1
            QuickFilename(I).Caption = QuickFilename(I + 1).Caption
        NEXT I
        UNLOAD QuickFilename(NumFiles)
        NumFiles = NumFiles - 1
    ELSE
        MSGBOX "The number is out-of-range", 0, "Input Error"
    END IF
END SUB

SUB DosCom_Click ()
        ' Invoke the DOSBAT batch file
        SHELL ("dosbat.bat " + DosBox.Text)
END SUB

SUB FindBox_Change ()
    FindCom.CHECKED = 0
END SUB

SUB FindCom_Click ()
    STATIC CurIndex AS LONG
    DIM Find AS STRING
    ' update the index to the current character
    CurIndex = TextBox.SelStart + 1
    ' store the text of the Find box in the variable Find
    Find = FindBox.Text
    ' is variable Find empty?
    IF Find = "" THEN EXIT SUB ' nothing to find
    ' locate the index of the next substring Find in the
    ' text box
    IF CaseSenseChk.Value = 1 THEN
        CurIndex = INSTR(CurIndex + 1, TextBox.Text, Find)
    ELSE
        CurIndex = INSTR(CurIndex + 1, UCASE$(TextBox.Text), UCASE$(Find))
    END IF
    ' found a match?
    IF CurIndex > 0 THEN
        ' Yes! Display the matching text as selected text
        TextBox.SelStart = CurIndex - 1
        TextBox.SelLength = LEN(Find)
        TextBox.SETFOCUS
        FindCom.CHECKED = TRUE
    ELSE
        ' No! Clear any selected text
        TextBox.SelStart = 0
        TextBox.SelLength = 0
        FindCom.CHECKED = FALSE
    END IF
END SUB

SUB FindFileCom_Click ()
    ' invoke the DOS dir command with the /s
```

Listing 13-1 Continued.

```
    ' subdirectory option and the /p paginated
    ' output option
    SHELL ("dosbat.bat dir " + FilenameBox.Text + " /s/p")
END SUB

SUB FindLbl_Change ()
    FindCom.CHECKED = FALSE
END SUB

SUB FindSelTextCom_Click ()
    IF TextBox.SelText <> "" THEN
        FindBox.Text = TextBox.SelText
    END IF
    FindCom_Click
END SUB

SUB Form_Load ()
    NumFiles = 0
    WindowState = 2
    ReadOnlyCom_Click
END SUB

SUB LoadCom_Click ()
    DIM F AS STRING
    DIM L AS STRING
    DIM NL AS STRING * 2
    ' obtain filename from the filename text box
    F = FilenameBox.Text
    ' if filename is an empty string, exit
    IF F = "" THEN EXIT SUB
    NL = CHR$(13) + CHR$(10)
    ' set error-handler
    ON LOCAL ERROR GOTO BadFile
    ' open the file
    OPEN F FOR INPUT AS 1
    F = "" ' clear variable to reuse it
    ' loop to read the text lines from the ASCII file
    DO WHILE NOT EOF(1)
        LINE INPUT #1, L
        F = F + L + NL ' append a new line
    LOOP
    TextBox.Text = F ' copy F into text box
    ' close the file
    CLOSE #1
    ' exit procedure
    EXIT SUB
' *********** Error-handler **********
BadFile:
    BEEP
    MSGBOX "Cannot open file " + F, 0, "File I/O Error"
    ON LOCAL ERROR GOTO 0
    RESUME EndOfSub
EndOfSub:
END SUB

SUB PasteCom_Click ()
    Screen.ActiveControl.SelText = Clipboard.GETTEXT()
END SUB

SUB QuickFilename_Click (Index AS INTEGER)
    FilenameBox.Text = QuickFilename(Index).Caption
    LoadCom_Click
END SUB

SUB QuitCom_Click ()
    END
END SUB

SUB ReadOnlyCom_Click ()
    SaveCom.Enabled = NOT SaveCom.Enabled
    ReadOnlyCom.CHECKED = NOT SaveCom.Enabled
END SUB
```

```
SUB SaveCom_Click ()
    IF (FilenameBox.Text <> "") AND (TextBox.Text <> "") THEN
        ON LOCAL ERROR GOTO BadSaveFilename
        OPEN FilenameBox.Text FOR OUTPUT AS #1
        ON LOCAL ERROR GOTO 0
        PRINT #1, TextBox.Text
        CLOSE #1
    END IF
    EXIT SUB

' ****************** Error-Handler ****************
BadSaveFilename:
    ON LOCAL ERROR GOTO 0
    MSGBOX "Cannot write to file " + FilenameBox.Text, 0, "File I/O Error"
    RESUME ExitSaveFile
ExitSaveFile:
END SUB

SUB ToggleLoadCom_Click ()
    LoadCom.Enabled = NOT LoadCom.Enabled
    ToggleLoadCom.CHECKED = NOT LoadCom.Enabled
END SUB

SUB ToggleOSCom_Click ()
    ' toggle the Visible property of the DosBox text box,
    ' the DosBtn button, the WinBtn button, and the DosLbl
    ' label
    DosBox.Visible = NOT DosBox.Visible
    DosCom.Visible = NOT DosCom.Visible
    WinCom.Visible = NOT WinCom.Visible
    DosLbl.Visible = NOT DosLbl.Visible
END SUB

SUB ViewCom_Click ()
    STATIC ViewClipboard AS INTEGER
    STATIC theFilename AS STRING
    STATIC theFileText AS STRING
    IF ViewClipboard THEN
        ViewClipboard = FALSE
        ViewCom.Caption = "View Clipboard"
        ClearCom.Enabled = TRUE
        AppendCom.Enabled = TRUE
        FilenameBox.Text = theFilename
        Clipboard.SETTEXT TextBox.Text
        TextBox.Text = theFileText
        theFileText = ""
    ELSE
        ViewClipboard = TRUE
        ViewCom.Caption = "View File"
        ClearCom.Enabled = FALSE
        AppendCom.Enabled = FALSE
        theFilename = FilenameBox.Text
        theFileText = TextBox.Text
        FilenameBox.Text = "Clipboard"
        TextBox.Text = Clipboard.GETTEXT()
    END IF
END SUB
```

Timer controls

Visual Basic offers the timer control. This is a special control in that it is always invisible at run time. The timer control is either disabled or enabled. When it is enabled, this control generates a Timer event at set time intervals.

Timer controls have a few properties, such as CtlName, Enabled, and Index, that are common with other controls. The Interval property is particular to the timer control. It can be set either at design time or at run time.

The general syntax for the Interval property is:

[form.]timer.Interval [=milliSeconds&]

The valid range of values for the Interval property is 1 to 65,535 milliseconds. The value of 60,000 represents one minute. Therefore, the maximum timer interval is just greater than one minute.

The timer control generates a Timer event every Interval milliseconds. The Timer event invokes the event-handling procedure of the associated timer control to perform the tasks you specify.

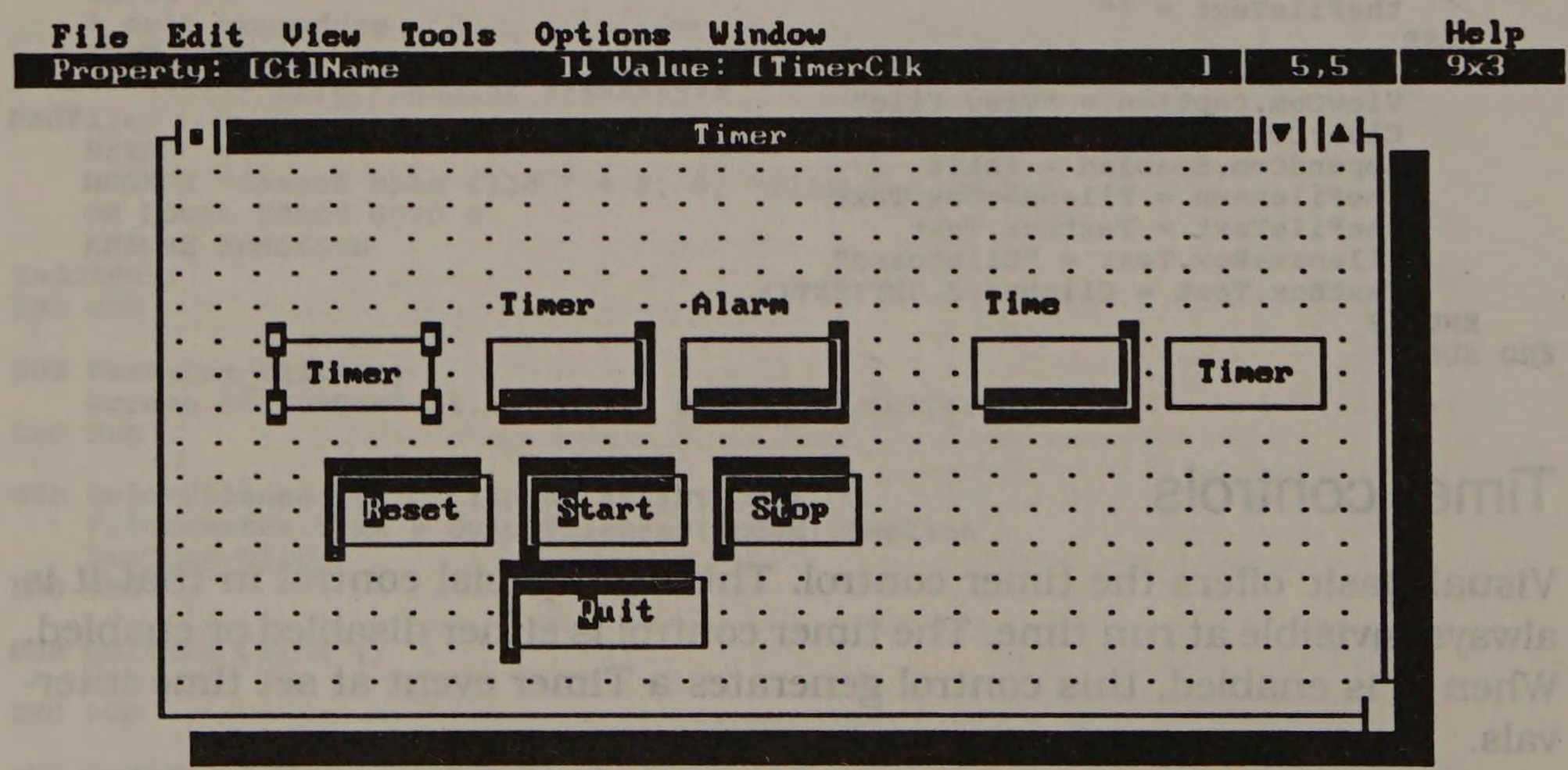
The following two steps are involved in using a timer control:

- Setting the Enabled property of the timer control to −1 (TRUE).
- Assigning a value to the Interval setting of the timer control.

To stop a timer control, set its Enabled property to 0 (FALSE).

Let's see how the timer control is put to work. The next program uses two timer controls, one as a timer and one to display a clock. Figure 13-6 shows the TIMER1.FRM form containing the various controls. The timer control is triggered by clicking on the Start button. The timer stops in one of three cases:

- You click on the Stop command button. This button simply stops the timer, but does not reset the display of the timer text box.
- You click on the Reset command button. The timer text box is reset to 00:00.
- The contents of the timer box match those of the alarm text box.



13-6 The TIMER1.FRM during design time.

You can type the number of minutes and seconds (in a mm:ss format) in the alarm text box to specify when the timer stops and displays a message box. You supply the text of the message box when you exit the alarm box (and the box is not empty)—a dialog box requests that you enter the message associated with the alarm. When the alarm is due (that is, when the text appearing in both the timer and alarm boxes is the same), the program pops a message box with the message you timed in earlier. This feature works even if you run the compiled program in minimized mode.

The program also has a text box that displays the current time. The time box is updated once a minute, regardless of whether or not you are running the timer.

Figures 13-7 to 13-9 contain the specifications for the TIMER1.MAK project file and the TIMER1.FRM file.

Application Name: Timers Demo
Application Code Name: TIMER1
Version: 1.0 Date Created: July 17, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	TIMER1.MAK
Form 1	TIMER1.FRM

13-7 General project specifications for the TIMER1 project.

Listing 13-2 contains the code attached to the TIMER1.FRM form. The following procedures are relevant to the timer controls:

The Form_Load procedure Performs the following tasks:

- Sets the NumSec (the number of seconds) variable to 0.
- Enables the TimeClk timer control and sets its Interval property to 60000 (one minute).
- Displays the current time in the time box.
- Resets the timer box.

The ResetBtn_Click Resets the timer by setting the Enabled property of the TimerClk control to FALSE. The procedure also sets the variable NumSec to zero and places 00:00 in the timer text box.

The StartBtn_Click procedure Triggers the timer by setting the TimerClk.Enabled property to TRUE. The procedure also assigns 987 milliseconds to the TimerClk.Interval property. This value is a little bit less than

Form # 1 Form Filename: TIMER1.FRM
Version: 1.0 Date: July 17, 1992

Control object type	Default CtlName	Purpose
Timer	Timer1	Implements a timer.
	Timer2	Displays an hh:mm clock.
Command Box	Command1	Exits the demo program.
	Command2	Resets the timer.
	Command3	Starts the timer.
	Command4	Stops the timer.
Text Box	Text1	Timer box.
	Text2	Alarm box.
	Text3	Clock box.
Label	Label1	Timer label.
	Label2	Alarm label.
	Label3	Clock label.

13-8 List of control objects for the TIMER1 project.

Application (Code) Name: TIMER1
Form # 1
Version: 1.0 Date: July 17, 1992

Original control name	Property	New setting
Form	Caption	Timer
Timer1	CtlName	TimerClk
Timer2	CtlName	TimeClk
Command1	CtlName	QuitBtn
	Caption	&Quit
Command2	CtlName	ResetBtn
	Caption	&Reset
Command3	CtlName	StartBtn
	Caption	&Start
Command4	CtlName	StopBtn
	Caption	&Stop
Text1	CtlName	TimerBox
	Text	(empty string)
Text2	CtlName	AlarmBox
	Text	(empty string)
Text3	CtlName	TimeBox
	Text	(empty string)
Label1	CtlName	TimerLbl
	Caption	Timer
Label2	CtlName	AlarmLbl
	Caption	Alarm
Label3	CtlName	TimeLbl
	Caption	Time

13-9 The modified control settings for the TIMER1 project.

1000, the "ideal" value to trigger the Timer event once every second. The difference accounts for the approximate time for code execution. You might need to fine tune this number further to accommodate the speed of execution on your system.

The StopBtn_Click procedure Disables the timer by simply assigning FALSE to the TimerClk.Enabled property.

The TimerClk_Timer procedure Displays the timer and performs the following tasks:

- Increments the NumSec variable that stores the time in seconds.
- Calculates the number of seconds and the number of minutes.
- Displays the mm:ss time in the timer text box.
- Compares the Text properties in the time and alarm text boxes. If these properties are equal, the timer stops and the alarm message is displayed. Two beeps are sounded before displaying the message box.
- The TimeClk_Timer displays the current time using the hh:mm format.
- The AlarmBox_LostFocus procedure enables you to type in the alarm message when you exit the alarm text box. If the alarm text box is empty, no action is taken.

Developing low-level keyboard handlers

Visual Basic offers the KeyDown and KeyUp events to allow you to detect the exact keys pressed. An earlier chapter presented the KeyPress event, which enabled you to process the ASCII characters being typed. By comparison, the KeyUp and KeyDown events provide you with information that is related to the *key* being pressed or released, not the ASCII code of the character it generates. The KeyUp and KeyDown events allow you to detect the following information:

- Whether or not the SHIFT, CTRL, and/or ALT key were also pressed.
- Detecting arrow keys and other cursor control keys.
- Separating the numeric keypad digits from the typewriter digits.

Using the KeyUp and KeyDown events enables you to perform a variety of tasks, such as:

- Providing on-line help, for example by pressing the F1 function.
- Performing additional processing on the Text property of a text box. For example, you might specify that a function key converts the characters in a text box into uppercase or lowercase. Using such a function key is an option that you can offer in your programs.

Listing 13-2 The code attached to the TIMER1.FRM form.

```
DECLARE SUB ResetBtn_Click ()
'$INCLUDE: 'CONSTANT.BI'
DIM SHARED NumSec AS LONG
DIM SHARED AlarmMessage AS STRING


SUB AlarmBox_LostFocus ()
    DIM S AS STRING
    IF AlarmBox.Text = "" THEN EXIT SUB
    S = INPUTBOX$("Enter alarm message", "Input", AlarmMessage)
    IF S <> "" THEN AlarmMessage = S
END SUB

SUB Form_Load ()
    NumSec = 0
    AlarmMessage = "Testing!"
    TimeClk.Enabled = TRUE
    TimeClk.Interval = 60000 ' once a minute
    TimeBox.Text = MID$(TIME$, 1, 5)
    ResetBtn_Click
END SUB

SUB QuitBtn_Click ()
      END
END SUB

SUB ResetBtn_Click ()
    NumSec = 0
    TimerBox.Text = "00:00"
    TimerClk.Enabled = FALSE
END SUB

SUB StartBtn_Click ()
    TimerClk.Enabled = TRUE
    TimerClk.Interval = 987
END SUB

SUB StopBtn_Click ()
      TimerClk.Enabled = FALSE
END SUB

SUB TimeBox_KeyPress (KeyAscii AS INTEGER)
    KeyAscii = 0
END SUB

SUB TimeClk_Timer ()
    TimeBox.Text = MID$(TIME$, 1, 5)
END SUB

SUB TimerBox_KeyPress (KeyAscii AS INTEGER)
    KeyAscii = 0
END SUB

SUB TimerClk_Timer ()
    DIM Secs AS INTEGER: DIM Min  AS INTEGER
    NumSec = NumSec + 1
```

```
    Secs = NumSec MOD 60
    Min = NumSec \ 60
    TimerBox.Text = FORMAT$(Min, "00") + ":" + FORMAT$(Secs, "00")
    IF AlarmBox.Text = TimerBox.Text THEN
        TimerClk.Enabled = FALSE
        BEEP:  BEEP
        MSGBOX AlarmMessage, 0, "Alarm!"
    END IF
END SUB
```

- Displaying preset text in a text box by pressing one or more function keys.
- Storing selected text or the contents of a text box in special strings.

The general form for the KeyUp and KeyDown event-handling procedure is shown next:

object_KeyUp (KeyCode AS INTEGER, Shift AS INTEGER)
object_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)

☞ The KeyCode parameter refers to the physical key that is pressed or released. Most keys produce multiple characters. For example, the typewrite key 5 also produces the % character when pressed with the SHIFT key. The KeyCode for the key producing the 5 and the % is the same.

The CONSTANT.BI file contains a set of KEY_XXXX global constants that code the various keys. Using these constants is definitely easier than trying to remember the hexadecimal key codes.

The Shift parameter plays the same role with the KeyUp and KeyDown events as it did with the MouseUp, MouseDown, and MouseMove events in chapter 13.

Let me illustrate using the KeyDown event to improve the WINEDIT1 program. The new program version, WINEDIT2, is identical in every way to WINEDIT1, except for the following special features:

- Pressing the SHIFT key with a function key (F1 to F10) stores the selected text of the edit text box in an element of a special array of strings.
- Pressing the CTRL key with a function key (F1 to F10) recalls the characters previously saved in the array of strings.

Listing 13-3 shows the additional code that I inserted in WINEDIT1 to produce program WINEDIT2. The general declarations section contains the declaration of a ten-member string array. This array stores various instances of the selected text (you can say that the array elements store snapshots of the selected text).

The TextBox_KeyDown procedure handles storing and recalling the characters. The IF-THEN-ELSEIF statement detects pressing the CTRL and the SHIFT keys along with a function key. The first tested condition com-

Listing 13-3 The additional code of WINDEDIT2.FRM.

```
' declare the strings that store the ten text blocks
DIM SHARED KeyStr(1 TO 10) AS STRING


SUB TextBox_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
  IF Shift = SHIFT_MASK THEN
    SELECT CASE KeyCode
      CASE KEY_F1: KeyStr(1) = TextBox.SelText
      CASE KEY_F2: KeyStr(2) = TextBox.SelText
      CASE KEY_F3: KeyStr(3) = TextBox.SelText
      CASE KEY_F4: KeyStr(4) = TextBox.SelText
      CASE KEY_F5: KeyStr(5) = TextBox.SelText
      CASE KEY_F6: KeyStr(6) = TextBox.SelText
      CASE KEY_F7: KeyStr(7) = TextBox.SelText
      CASE KEY_F8: KeyStr(8) = TextBox.SelText
      CASE KEY_F9: KeyStr(9) = TextBox.SelText
      CASE KEY_F10: KeyStr(10) = TextBox.SelText
    END SELECT
  ELSEIF Shift = CTRL_MASK THEN
    SELECT CASE KeyCode
      CASE KEY_F1: TextBox.SelText = KeyStr(1)
      CASE KEY_F2: TextBox.SelText = KeyStr(2)
      CASE KEY_F3: TextBox.SelText = KeyStr(3)
      CASE KEY_F4: TextBox.SelText = KeyStr(4)
      CASE KEY_F5: TextBox.SelText = KeyStr(5)
      CASE KEY_F6: TextBox.SelText = KeyStr(6)
      CASE KEY_F7: TextBox.SelText = KeyStr(7)
      CASE KEY_F8: TextBox.SelText = KeyStr(8)
      CASE KEY_F9: TextBox.SelText = KeyStr(9)
      CASE KEY_F10: TextBox.SelText = KeyStr(10)
    END SELECT
  END IF
END SUB
```

pares the Shift argument with the global constant SHIFT_MASK. This condition is true when the SHIFT key is pressed along with a function key. If the condition is true, a SELECT-CASE statement is used to store the selected text in the proper element of array KeyStr.

The ELSEIF clause compares the Shift argument with the global constant CTRL_MASK. This condition is true when the CTRL key is pressed simultaneously with a function key. If the condition is true, a SELECT-CASE statement is used to recall the characters from the proper element of array KeyStr.

Run the WINEDIT2 program and use this new feature. Now you can emulate text macros with the text editor.

14
Run time errors

The ideal program is one that does not encounter any errors. The reality of programming practices involves dealing with various levels of errors. First, there is the syntax error that is generated by typos and the wrong use of procedures or functions. Such errors are caught by the syntax checker or the compiler. The second level of errors is the logical program error when the program does what you are telling it to do and not what you really want it to do. Such errors are detected by testing the operations of the program. These two types of errors are related to the design time. A third kind of error creeps up at run time when invalid arguments (I am using this term in a broad sense) are supplied in a statement.

This chapter briefly looks at the error-handling techniques offered by Visual Basic. Before you read on, I want to point out that the error-handling facilities of Visual Basic are practically identical to those of QuickBasic. Visual Basic offers the new level of error trapping that works within procedures.

Run time errors can appear in just about every aspect of your program. Every task executed by even one block of statements can go wrong (Murphy's law). Therefore, the examples presented here are for run time errors only.

Defensive programming techniques

There was a car commercial on TV that said, "The best way to survive a collision is to avoid having one." This is the basic philosophy of defensive programming techniques—avoid the occurrence of a run time error. Defensive programming employs code that checks for possible error-generating conditions to avoid the occurrence. This preemptive approach to run time error-

handling (or error-avoiding, if you prefer) is optional in Visual Basic and other Microsoft Basic implementations. It is interesting to point out that prominent languages like Pascal, C, and Modula-2 rely on defensive programming techniques to avoid run time errors.

An example of defensive programming is shown in the next Function procedure that returns a −1 if the file argument exists, and 0 when otherwise. Attempting to open a nonexistent file for input generates a run time error. A function like FileExists enables you to avoid that type of run time error. The function is shown below:

```
FUNCTION FileExists (Filename AS String) AS INTEGER
    IF DIR$(Filename) THEN
        FileExists = 0 ' FALSE
    ELSE
        FileExists = -1 ' TRUE
    END IF
END IF
```

The FileExists function employs the predefined DIR$() to return the first name of the file that matches the wildcard filename argument; in the case of the FileExists function, the filename itself is used as the search wildcard filename. If the DIR$ function returns an empty string no matching file is found, because the specified file does not exists. Otherwise, the DIR$ returns a nonempty string, because the sought file does exist.

The function FileExists accomplishes its task without using the technique of attempting to open the sought file for input to see whether it exists.

Visual Basic error handling

Visual Basic offers the ON ERROR and RESUME statements to trap and handle run time errors. The general syntax for the ON ERROR statement is:

```
ON [LOCAL] ERROR { GOTO { line | label } | RESUME NEXT ¦ GOTO 0 }
```

The LOCAL keyword is permitted in procedure-level error-handling for the sake of compatibility with other Microsoft Basic implementation. The GOTO { line | label } clause directs program execution to the line containing the specified line number of label. The line or label must reside in the same procedure; otherwise, Visual Basic emits an error message. The RESUME NEXT clause indicates that the program should resume at the next statement (the error is practically ignored). The GOTO 0 clause disables any active error-handling.

When the program execution is directed to the error-handling statements, you typically use the RESUME statement to determine how to resume the program. The general syntax for the Resume statement is:

```
RESUME { [0] ¦ NEXT ¦ line }
```

The RESUME 0 statement causes the program execution to continue with the offending statement, or the most recently executed call out of the procedure that contains the error-handler.

The RESUME NEXT statement results in resuming the program execution with the statement that directly follows the offending statement, or with the statement that follows the most recently executed call out of the procedure that contains the error-handler.

The RESUME line statement directs program execution to a specific local label.

Visual Basic reports additional error when misapplying the RESUME statement in the following cases:

- A RESUME statement is encountered when there is no error condition to handle. In other words, you must not allow program execution to run into a RESUME statement when no error is raised.
- The RESUME line statement points to a label outside the current procedure.
- Executing an END, END SUB, or END FUNCTION before a RESUME statement is encountered, when an error is raised by the run time system. This means that you cannot sweep a run time error under the rug, so to speak!

Visual Basic offers the following functions that enable you to examine the type of the error that occurred:

- The ERR function returns an integer which represents the run time error code for the error that just occurred.
- The ERRL function returns an integer that indicates the line number of the offending statement, or the closest line number before that statement.
- The ERROR$ function returns the message that describes the most recent error or the error of a given code. The general syntax of the ERROR$ function is:

```
[stringExpression$ =] ERROR$([errorCode%])
```

When the errorCode% argument is omitted, the ERROR$ function describes the most recent error. This is the same message you obtain from the expression ERROR$(ERR).

Visual Basic also offers the ERROR statement to simulate a run time error. The general syntax for the ERROR statement is:

```
ERROR errorCode%
```

The range of values for the errorCode% argument is 1 to 32767. When the errorCode% matches a Visual Basic error code, that error is simulated. To

use your own error code, Microsoft recommends that you start with 32767 and work your way down. This strategy ensures that you have enough numbers to simulate your own errors and not overlap with the numbers reserved by Visual Basic. The ERROR$ function returns the string "User-defined error" for error codes that are not used by Visual Basic.

Visual Basic allows you to turn off an error-handler by executing an ON ERROR GOTO 0 statement.

The types of run time errors

This section describes various ways of handling run time errors. This list is by no means comprehensive. Instead, it includes the most frequent error handling strategies.

The fatal error type This is the type of error that cannot be recovered without at least rebooting or even changing machines. In this case an ON ERROR GOTO RESUME directs the program execution to the error-handling section that displays a message and then ends the program. A general form of a procedure that handles fatal errors is shown below:

```
SUB CriticalProcedure ()
    ' statements that do not generate error
    ON ERROR RESUME FatalError
    . . .
    ' error generating statement(s)
    . . .
    EXIT SUB
FatalError:
    MSGBOX FatalErrorMessage$, 0, "Fatal Error!"
        END
END SUB
```

The abandon-error type This is the kind of error that causes you to exit the host procedure. I have used this type of error-handler with procedures that perform file I/O. Frequently, file I/O errors are caused by a bad filename. The strategy used directs the program execution to the error-handling statements, as shown next:

```
SUB DoFileIO (Filename AS String)
    ' statements that do not generate error
    ON ERROR GOTO FileIOError
    . . .
    OPEN Filename FOR Input AS #1
    . . .
    EXIT SUB
FileIOError:
    MSGBOX ErrorMessage$, 0, "File I/O Error"
        RESUME ExitDoFileIO
ExitDoFileIO:
END SUB
```

The error-handling statements display an error message and then resume to a label placed just before the END SUB.

The abandon-error type can also be used to implement a version of the FileExists function, as shown below:

```
FUNCTION FileExists (Filename AS String) AS INTEGER
    ON ERROR GOTO FileNotFound
    OPEN Filename FOR INPUT AS #1
    CLOSE #1
    FileExists = -1 ' TRUE
    EXIT FUNCTION
FileNotFound:
    RESUME NextLine
NextLine:
    FileExists = 0 ' FALSE
END FUNCTION
```

The ignore-error type This error-handling strategy assumes that you can resume executing the next statement. For example, you need to rename a file with the .BAK extension name. However, you need to first make sure that any possible .BAK file with the same name is erased. The RESUME NEXT clause, shown below, permits you to delete the older .BAK first:

```
ON ERROR RESUME NEXT
KILL filename$ + ".BAK"
ON ERROR GOTO 0
```

If the older .BAK file does not exist, the error trap enables you to simply resume program execution at the next statement.

The retry type This kind of error is typically associated with trying to read a file from a disk whose door is open. The error handling statements display a message asking you to make sure that the disk drive door is closed before you try reading the file again. The general form of a procedure containing this kind of error-handling scheme is shown next:

```
SUB DoFileIO (Filename AS String)
    ' statements that do not generate error
    ON ERROR GOTO FileIOError
    . . .
    OPEN Filename FOR INPUT AS #1
    . . .
    EXIT SUB
FileIOError:
    IF ERR = Err_DiskNotReady THEN
        MSGBOX "Disk drive door is open, please close it", 0, "Error"
        RESUME 0
    ELSE
        MSGBOX ErrorMessage$, 0, "File I/O Error"
        RESUME ExitDoFileIO
    END IF
```

```
ExitDoFileIO:
END SUB
```

The ELSE clause of the IF statement in the error-handling section deals with other file I/O errors.

The limited-retry method This scheme is a modified version of the last one. The basic philosophy is that there should be a limit to certain types of errors. A maximum limit is specified using a constant (either local or global) along with an error occurrence counter variable. The general form is shown here:

```
SUB DoFileIO (Filename AS String)
    CONST MAX_ERRORS = 3
    DIM ErrorCount
    ErrorCount = 0
    ' statements that do not generate error
    ON ERROR GOTO FileIOError
    . . .
    OPEN Filename FOR INPUT AS #1
    . . .
    EXIT SUB
FileIOError:
    IF ERR = Err_DiskNotReady THEN
        ErrorCount = ErrorCount + 1
        IF ErrorCount < MAX_ERRORS THEN
            MSGBOX "Disk drive door is open, please close it", 0,
            "Error"
            RESUME 0
        ELSE
            MSGBOX "Drive is considered invalid", 0, "Bad Error!"
            RESUME ExitDoFileIO
    ELSE
        MSGBOX ErrorMessage$, 0, "File I/O Error"
        RESUME ExitDoFileIO
    END IF
ExitDoFileIO:
END SUB
```

The THEN clause of the first IF statement increments the ErrorCount variable and compares its value with MAX_ERRORS. While the number of error occurrences is below the limit tolerated, the code uses a RESUME 0 to retry executing the offending statement. Beyond that limit, the code performs a RESUME ExitDoFileIO to exit the procedure.

The retry-with-new-data type This error-handling approach looks for alternate information that will not raise the error condition when the offending statement is executed again. The new data is obtained either by prompting the user (typically using the INPUTBOX$ function) or by retrieving data

from an array or a control. The following general form shows the first case (user input):

```
SUB DoFileIO (Filename AS String)
    DIM S AS String
    ' statements that do not generate error
    ON ERROR GOTO FileIOError
    . . .
    OPEN Filename FOR INPUT AS #1
    . . .
    EXIT SUB
FileIOError:
    S = INPUTBOX$(ErrorMessage$, "File I/O Error", Filename)
    IF S <> "" THEN ' user pressed the OK button
        Filename = S ' update filename
        RESUME 0 ' retry file I/O statement
ELSE ' user pressed the Cancel button
        RESUME ExitDoFileIO ' exit
    END IF
ExitDoFileIO:
END SUB
```

When you use the INPUTBOX$, you should take into consideration that the user might press the Cancel button. This action is interpreted as meaning the user does not wish to supply new data (for whatever reason). Consequently, the procedure is exited.

The delayed-error-handling type This approach stores the error code in a variable in order to deal with the error later. Unless this approach makes better sense or is far more suitable than the previous ones, I do not recommend it.

15

Working with the file system controls

Professional looking DOS applications, including Visual Basic, typically employ filename dialog boxes when you want to load a file or save a file under a new name. If you regularly use Microsoft Windows, then you have grown accustomed to this kind of dialog box, because it enables us to navigate through any drive and any directory.

The Visual Basic designers have elected to include control objects that permit Visual Basic applications to offer the above type of filename dialog boxes. These controls are the drive list box, the directory list box, and the file list box. This chapter covers the relevant properties and events of the drive list box, the directory list box, and the file list box. The text then continues on to guide you through using these controls to implement a directory tree browser, and a general-purpose filename dialog box.

The drive list box

The drive list box is a drop-down combo list box that returns the current drive and switches between the disk drives at run time. The following are properties relevant to selecting a current drive.

The Drive property Returns and assigns the selected drive at run time. This property is not available at design time. The general syntax for the Drive property is:

[form.]driveListBox.Drive [=driveName$]

The valid range of values for the Drive setting includes all the drives that are online (physical or electronic). By default, the Drive setting is assigned the

current drive. The Drive property returns the drive information using one of the following formats:

- "a:", "b:" and so on for floppy drives.
- "c:[volume id]" for hard disks.

When you write to a new drive you should observe the following rules:

- The first character of the driveName$ string is the only significant part of the assigned string. The assignment is not case-sensitive.
- When the Drive property is changed, a Change event occurs.
- Assigning a new Drive setting causes the regeneration of the drive list.
- Assigning a nonexistent drive results in a run time error.

The ListCount, List, and ListIndex properties These properties are similar to the ones in the list box and combo list box controls presented earlier in this book. The difference between the drive list box and the other list box controls is that the drive's properties are read-only. The ListCount property specifies the number data items in the drive list box. The ListIndex points to the currently selected data item. The values of the ListIndex range from 0 to ListCount – 1. The List(I%) property enables you to select the I%'th list item.

The change event is the event that is relevant to changing the current drive. This event can be used to update the directory list box (more about this later in this chapter).

The directory list box

The directory list box shows the directories and the current path at run time. The directory list box displays the ancestors directories and subdirectories of the current directory. When you select the root directory, the control shows all of the immediate subdirectories in the current drive. The properties relevant to selecting a current directory are:

The Path property Returns and assigns the selected directory at run time. This property is not available at design time. The general syntax for the Path property is:

```
[form.]fileListBox.Path [=pathName$]
```

The default Path setting is the current directory. The Path property retrieves the current directory. Assigning a new setting to the Path property selects a new directory. This has the same effect as the ChDir statement. The assigned directory can be a partial path or even a drive name (without the

colon). In the latter case, the Path setting changes the current drive and simultaneously selects the current directory of that drive.

Assigning a new Path setting results in the occurrence of a Change event for the directory list box and a PathChange event for the accompanying file list boxes.

The ListCount, List, and ListIndex properties These properties are similar to the ones in the drive list box.

The Change event is the event that is relevant to changing the current directory. This event can be used to update the file list box (more about this later in this chapter).

The file list box

The file list box is a drop-down combo list box that selects a file to be opened, saved, deleted, copied, renamed, or processed in other ways. The file list box properties that are relevant to selecting a file are:

The Pattern property Defines which filenames are displayed in the file list box. The general syntax for the Pattern property is:

```
[form.]fileListBox.Pattern [=filenameWildcard$]
```

The default Pattern setting is *.* that shows all of the files. The Pattern property returns the current filename pattern. To change the list of files, assign a new setting to the Pattern property. Typical pattern settings include wildcards, such as *.*, *.BAS, *.EXE, and *.TXT, to name just a few. When you change the Pattern setting a PatternChange event occurs. Visual Basic allows you to specify multiple wildcards in the Pattern property. These multiple wildcards must be separated by semicolons. An example of a valid multiple wildcard pattern is the string "*.EXE; *.COM; *.BAT". This pattern requests the .EXE, .COM. and .BAT files to appear in the file list box control.

The file attribute properties These Boolean properties, which can be set or queried, specify files that appear in the list box based on the matching file attributes. These properties are the Archive, Hidden, Normal, ReadOnly, and System properties. The general syntax for these properties is shown below:

```
[form.]fileListBox.Archive [=Boolean%]
[form.]fileListBox.Hidden [=Boolean%]
[form.]fileListBox.Normal [=Boolean%]
[form.]fileListBox.ReadOnly [=Boolean%]
[form.]fileListBox.System [=Boolean%]
```

The default setting for the Archive, Normal, and ReadOnly properties is −1 (TRUE). The default setting for the other properties is 0 (FALSE).

☞ The file list box displays a file that meets the following conditions:

- The file fits the filename Pattern setting, such as *.*, *.EXE, or *.TXT.
- All of the file attributes correspond to file attribute settings that are equal to −1 (TRUE).
- The file attribute properties act as an additional file display filter together with the Pattern property.
- The ListCount, List, and ListIndex properties. These properties are similar to the ones in the drive list box. The selected file is given by the expression fileListBox.List(fileListBox.ListIndex).
- The Filename property returns the file selected in the file list box. This property also allows you to assign a new drive, path, and/or pattern. The general syntax for the Filename property is:

  [form.]fileListBox.Filename [=filename$]

When the program starts running, the setting of the Filename property is an empty string, because there is no file selected. Using the Filename property is the shortcut for using the List and ListIndex properties.

When you assign new Filename setting you should take note of the following rules:

- If the new setting contains a drive, path, or pattern the Drive property (of a drive list box), Path property (of a directory list box), and Pattern property are automatically updated.
- If the new Filename setting specifies the name of an existing file, that file is displayed and also selected.
- The new setting may generate PathChange, PatternChange, and DblClick (when you select an existing file).

Next, I present two programs that show how the various properties of the drive, directory, and file list boxes work together.

The directory tree viewer

The first demonstration program is a simple utility that displays the files in the current directory. The Pattern and file attribute properties specify the files that appear in the file list box. In addition, I implemented a secondary feature that automatically selects a predefined file.

Figures 15-1 through 15-3 contain the specifications for the TREEVU1.MAK project file and the TREEVU1.FRM form file. Figure 15-4 shows a sample session with the directory tree viewer.

Listing 15-1 contains the code attached to the TREEVU1.FRM form. The relevant procedures are as follows:

Application Name: Directory Tree Viewer	
Application Code Name: TREEVU1	
Version: 1.0	Date Created: July 19, 1992
Programmer(s): Namir Clement Shammas	
	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	TREEVU1.MAK
Form 1	TREEVU1.FRM

15-1 The basic specifications for the TREEVU1.MAK project file.

Form # 1	Form Filename: TREEVU1.FRM
Version: 1.0	Date: July 19, 1992

Control object type	Default CtlName	Purpose
Drive List Box	Drive1	Shows and selects the current drive.
Directory List	Dir1	Shows and selects the current directory.
File List Box	File1	Shows a group of files.
Frame	Frame1	Contains the file attribute check boxes.
Check Box	Check1	Toggles the Archive property.
	Check2	Toggles the Hidden property.
	Check3	Toggles the ReadOnly property.
	Check4	Toggles the Normal property.
	Check5	Toggles the System property.
Text Box	Text1	Specifies the filename pattern.
	Text2	Specifies the selected file.
Command Box	Command1	Exits the program.
	Command2	Finds a matching file.
Label	Label1	Label for filename pattern.
	Label2	Label for selected file.
	Label3	Label for drive list box.
	Label4	Label for directory list box.
	Label5	Label for file list box.

15-2 The list of controls in the TREEVU1.FRM form.

The Form_Load procedure Initializes the form and its controls when loaded. Of interest are the following initialization tasks:

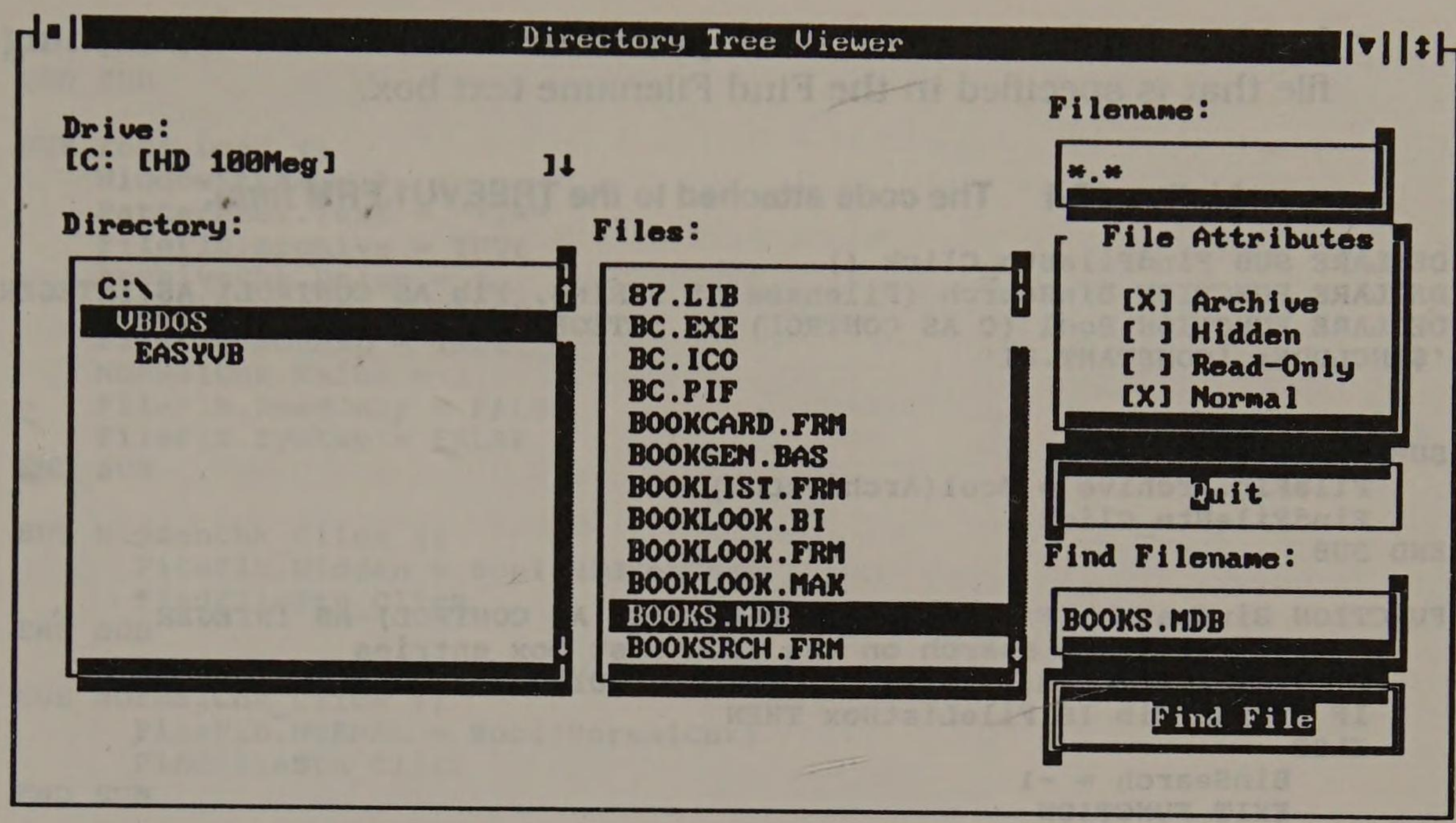
- Assigns *.* to the Text property of the pattern text box.
- Assigns the global constant TRUE to the Archive and Normal properties of the FileFlb file list box.

Application (Code) Name: TREEVU1
Form # 1
Version: 1.0 Date: July 19, 1992

Original control name	Property	New setting
Form	Caption	Directory Tree Viewer
Drive1	CtlName	DiskDrv
Dir1	CtlName	TreeDir
File1	CtlName	FileFlb
Frame1	CtlName	FileAttrFrm
	Caption	" File Attributes" (no quotes)
Check1	CtlName	ArhiveChk
	Caption	Archive
Check2	CtlName	HiddenChk
	Caption	Hidden
Check1	CtlName	ReadOnlyChk
	Caption	Read-Only
Check1	CtlName	NormalChk
	Caption	Normal
Check1	CtlName	SystemChk
	Caption	System
Text1	CtlName	PatternBox
	Text	(empty string)
Text2	CtlName	FindFileBox
	Text	(empty string)
Command1	CtlName	QuitBtn
	Caption	&Quit
	Cancel	TRUE
Command2	CtlName	FindFileBtn
	Caption	&Find File
Label1	CtlName	PatternLbl
	Caption	Filename:
Label2	CtlName	FindFileLbl
	Caption	Find Filename:
Label3	CtlName	DriveLbl
	Caption	Drive:
Label4	CtlName	DirectoryLbl
	Caption	Directory:
Label5	CtlName	FilesLbl
	Caption	Files:

15-3 The customized settings of the TREEVU1.FRM form.

- Sets the Value property of the Archive and Normal check boxes to 1. These controls appear checked when the program starts running.
- Assigns the global constant FALSE to the Hidden, ReadOnly, and System properties of the FileFlb file list box.



15-4 A sample session with the TREEVU1 program.

The DiskDrv_Change procedure Updates the Path setting of the directory list box with the new Drive setting of the drive list box. This synchronizes the directory list box with the drive list box.

The TreeDir_Change procedure Updates the Path setting of the file list box with the new Path setting of the directory list box. The procedure also invokes the FindFileBtn_Click procedure to search for the target file in the updated file list box.

The FileFlb_PatternChange procedure Calls the FindFileBtn_Click procedure when the pattern is updated.

The Bool function Returns TRUE if the Value setting of its control-typed parameter is 1. If the setting is 0, the function returns FALSE.

The family procedures Handle the click events for the check boxes. These procedures set the file attribute properties based on the Value setting of the corresponding check box. The Bool function obtains a TRUE or FALSE value from the check box argument.

The PatternBox_LostFocus procedure Essentially updates the Filename property of the file list box. This procedure performs the following tasks:

- Assigns *.* to the Text setting if the pattern text box is empty.
- Assigns the Text property of the pattern text box to the Filename property. You can replace the Filename property with the Pattern property. The program works fine until you specify a pattern that includes a drive; then, Visual Basic generates a run time error.

- Invokes the FindFileBtn_Click procedure to search for a matching file that is specified in the Find Filename text box.

Listing 15-1 The code attached to the TREEVU1.FRM form.

```
DECLARE SUB FindFileBtn_Click ()
DECLARE FUNCTION BinSearch (Filename AS STRING, Flb AS CONTROL) AS INTEGER
DECLARE FUNCTION Bool (C AS CONTROL) AS INTEGER
'$INCLUDE: 'CONSTANT.BI'


SUB ArchiveChk_Click ()
    FileFlb.Archive = Bool(ArchiveChk)
    FindFileBtn_Click
END SUB

FUNCTION BinSearch (Filename AS STRING, Flb AS CONTROL) AS INTEGER
    ' perform binary search on the file list box entries
    DIM Lo AS INTEGER: DIM Hi  AS INTEGER: DIM Median  AS INTEGER
    IF TYPEOF Flb IS FileListBox THEN
    ELSE
        BinSearch = -1
        EXIT FUNCTION
    END IF
    Lo = 0
    Hi = Flb.ListCount - 1
    DO
        Median = (Lo + Hi) \ 2
        IF Filename < Flb.List(Median) THEN
            Hi = Median - 1
        ELSE
            Lo = Median + 1
        END IF
    LOOP UNTIL (Filename = Flb.List(Median)) OR (Lo > Hi)
    IF Filename = Flb.List(Median) THEN
        BinSearch = Median
    ELSE
        BinSearch = -1
    END IF
END FUNCTION

FUNCTION Bool (C AS CONTROL) AS INTEGER
    Bool = -1 * C.Value
END FUNCTION

SUB DiskDrv_Change ()
      ' update file tree path when the disk is changed
    TreeDir.Path = DiskDrv.Drive
END SUB

SUB FileFlb_PatternChange ()
    FindFileBtn_Click
END SUB

SUB FindFileBox_KeyPress (KeyAscii AS INTEGER)
    KeyAscii = ASC(UCASE$(CHR$(KeyAscii)))
END SUB

SUB FindFileBtn_Click ()
    DIM I AS INTEGER
    DIM S AS STRING
    IF FindFileBox.Text = "" THEN EXIT SUB
    S = FindFileBox.Text
    I = BinSearch(S, FileFlb)
    IF I > -1 THEN
        FileFlb.ListIndex = I
```

```
    END IF
END SUB

SUB Form_Load ()
    WindowState = 2
    PatternBox.Text = "*.*"
    FileFlb.Archive = TRUE
    ArchiveChk.Value = 1
    FileFlb.Hidden = FALSE
    FileFlb.NORMAL = TRUE
    NormalChk.Value = 1
    FileFlb.ReadOnly = FALSE
    FileFlb.System = FALSE
END SUB

SUB HiddenChk_Click ()
      FileFlb.Hidden = Bool(HiddenChk)
      FindFileBtn_Click
END SUB

SUB NormalChk_Click ()
      FileFlb.NORMAL = Bool(NormalChk)
      FindFileBtn_Click
END SUB

SUB PatternBox_KeyPress (KeyAscii AS INTEGER)
    KeyAscii = ASC(LCASE$(CHR$(KeyAscii)))
END SUB

SUB PatternBox_LostFocus ()
    DIM I AS INTEGER
    IF PatternBox.Text = "" THEN PatternBox.Text = "*.*"
    FileFlb.Filename = PatternBox.Text
    FindFileBtn_Click
END SUB
SUB QuitBtn_Click ()
      END
END SUB

SUB ReadOnlyChk_Click ()
      FileFlb.ReadOnly = Bool(ReadOnlyChk)
      FindFileBtn_Click
END SUB

SUB SystemChk_Click ()
      FileFlb.System = Bool(SystemChk)
      FindFileBtn_Click
END SUB

SUB TreeDir_Change ()
      FileFlb.Path = TreeDir.Path
      FindFileBtn_Click
END SUB
```

The FindFileBtn_Click procedure Searches for the file list box member that matches the filename in the Find Filename text box. If the latter text box is empty the search is aborted. The BinSearch function is invoked to obtain the index of the matching item file list box. If no match is found the BinSearch function returns −1. When a match is found the ListIndex of the file list box is set to the index of the matching item. This causes that item to be selected.

The BinSearch function Searches for the Filename argument in the Flb file list box argument. I wrote this function as a general-purpose routine. The function uses the efficient binary search method algorithm that is applied to the ordered members of the file list box. The function returns the index of the matching list item, or −1 when no match is found.

Run the directory tree viewer program and experiment with selecting different drives, directories, filename patterns, and file attributes. Once you get a feel for the basic program features, type in a filename in the Find Filename text box.

I suggest that you select the name of a file that is present in more than one directory. Click on the Find File button. If the sought file is in the current directory, the program selects it. Now move to other directories that contain a file that matches the file specified in the Find Filename text box. As soon as the file list appears, the program zooms in on the matching file and selects it.

The text editor

Chapter 13 presented two versions of a simple menu-driven text editor. The Load and Add menu options prompted you for the name of a file using a simple dialog box. These programs lack the ability to show you a file dialog box that enables you to choose the file you want without guessing its correct name and/or path.

This section presents a new version of the text editor that uses a dialog box with a drive, directory, and file list boxes. The program uses two forms. The first form shows the same interface of the WINDEDIT1 and WINEDIT2 programs. The second form is shown in FIG. 15-5. That form contains the dialog box controls.

The new version of the text editor, WINEDIT3, uses the modal file dialog box when you invoke the Load and Add options of the File menu.

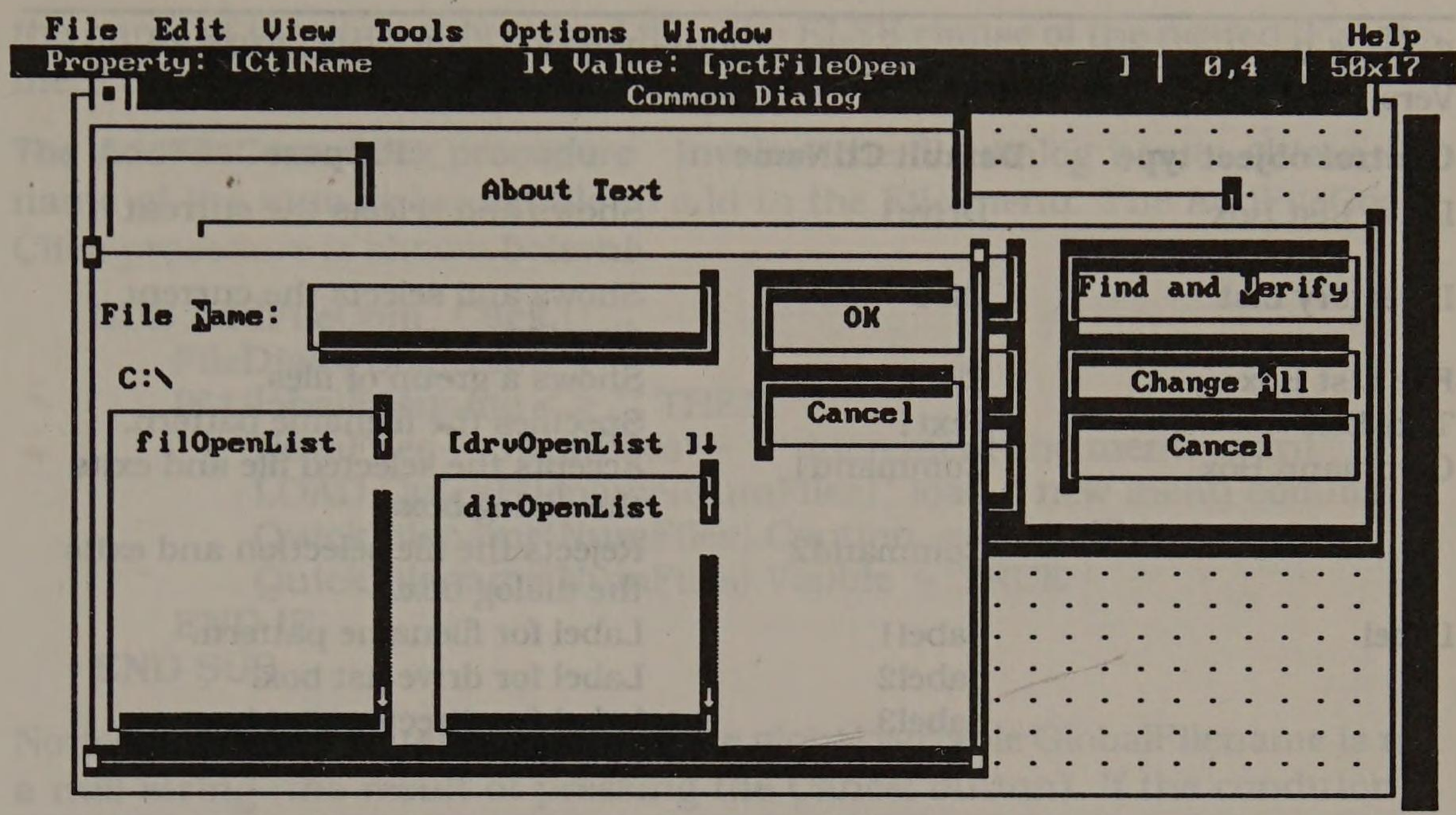
Figure 15-6 contains the specifications for the WINEDIT3.MAK project file. The specifications for the WINEDIT3.FRM form file and its menu structure is the same as those shown in Figures 14-3 through 14-5. Figures 15-7 and 15-8 contain the specifications for the FILEDLG.FRM form file.

The program uses the common shared variable GlobalFilename in both forms to report the name of the selected file in form FILEDLG. If you make no selection, by pressing the Cancel button, the FILEDLG form returns an empty string in variable GlobalFilename.

Listing 15-2 contains the code attached to the WINEDIT3.FRM form. The updated code portions for this new version are:

The UseFileDialogBox variable A Boolean flag used to decide whether or not to invoke the file dialog box.

The Form_Load procedure Initializes the UseFileDialogBox variable with the global constant TRUE.



15-5 The FILEDLG.FRM interface.

Application Name: Window Text Editor (version 3)	
Application Code Name: WINEDIT3	
Version: 1.0	Date Created: July 19, 1992
Programmer(s): Namir Clement Shammas	
	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	WINEDIT3.MAK
Form 1	WINEDIT3.FRM
Form 2	FILEDLG.FRM

15-6 The general specifications for the WINEDIT3.MAK project file.

The LoadCom_Click procedure Now includes the following If-Then statement that decides whether or not to load and invoke the file dialog box form:

```
IF UseFileDialogBox THEN
    FileDialogBox.SHOW 1
    IF GlobalFilename <> "" THEN
        FilenameBox.Text = GlobalFilename
    ELSE
        EXIT SUB
    END IF
END IF
```

A nested IF statement decides whether or not the global variable GlobalFilename is assigned to the filename text box. This enables you to preserve

Form # 2 Form Filename: FILEDLG.FRM
Version: 1.0 Date: July 19, 1992

Control object type	Default CtlName	Purpose
Drive List Box	Drive1	Shows and selects the current drive.
Directory List	Dir1	Shows and selects the current directory.
File List Box	File1	Shows a group of files.
Text Box	Text1	Specifies the filename pattern.
Command Box	Command1	Accepts the selected file and exits the dialog box.
	Command2	Rejects the file selection and exits the dialog box.
Label	Label1	Label for filename pattern.
	Label2	Label for drive list box.
	Label3	Label for directory list box.
	Label4	Label for file list box.

15-7 The list of controls of the FILEDLG.FRM form.

Application (Code) Name: WINEDIT3
Form # 2
Version: 1.0 Date: July 19, 1992

Original control name	Property	New setting
Form	Caption	File Dialog Box
	Form-Name	FileDialogBox
Drive1	CtlName	DiskDrv
Dir1	CtlName	TreeDir
File1	CtlName	FileFlb
Text1	CtlName	FilenameBox
	Text	(empty string)
Command1	CtlName	OKBtn
	Caption	&OK
Command2	CtlName	CancelBtn
	Caption	&Cancel
	Cancel	TRUE
Label1	CtlName	FilenameLbl
	Caption	Filename:
Label2	CtlName	DriveLbl
	Caption	Drive:
Label3	CtlName	DirectoryLbl
	Caption	Directory:
Label4	CtlName	FilesLbl
	Caption	Files:

15-8 The customized settings of form FILEDLG.FRM.

the name of the currently viewed file. The ELSE clause of the nested IF statement causes program execution to exit the LoadCom_Click procedure.

The AddFileCom_Click procedure Invokes the file dialog box to obtain the name of the quickly-loaded file to add in the File menu. The AddFileCom_Click procedure is shown below:

```
SUB AddFileCom_Click ()
    FileDialogBox.Show 1
    IF GlobalFilename <> "" THEN
        NumFiles = NumFiles + 1 ' increment the menu count
        LOAD QuickFilename(NumFiles) ' load a new menu command
        QuickFilename(NumFiles).Caption = GlobalFilename
        QuickFilename(NumFiles).Visible = TRUE
    END IF
END SUB
```

Notice that an IF statement tests if the global variable GlobalFilename is not a null string (the result of pressing the Cancel button). If the condition is true, the selected filename is added to the list of QuickFilename controls.

The QuickFilenameCom_Click procedure Assigns the FALSE constant to the variable UseFileDialogBox before calling LoadCom_Click. The procedure assigns the constant TRUE to the variable UseFileDialogBox after calling LoadCom_Click, restoring the normal value of that variable. If the first assignment to the variable UseFileDialogBox is omitted, the procedure ends up invoking the file dialog box. This task defeats the purpose of having the quickly-loaded files in the first place.

Listing 15-3 shows the code attached to the FILEDLG.FRM form. The procedures in that form are:

The Form_Load procedure Initializes the file attribute properties, setting the Archive and Normal properties to TRUE, and all of the other file attribute properties to FALSE. The string *.* is assigned to both the Pattern property of the file list box and the Text property of the filename text box.

The CancelBtn_Click procedure Implements a cancel request. It assigns a null string to the global variable GlobalFilename and unloads the file dialog form and returns the control to the main form.

The OKBtn_Click procedure Accepts the selected file and assigns it (along with the file's path) to the global variable GlobalFilename. The procedure then unloads the file dialog form and returns the control to the main form.

The FileFlb_DblClick procedure Implements a shortcut for accepting the selected file. The procedure simply invokes OKBtn_Click, which does the rest of the work.

The DiskDrv_Change procedure Updates the Path setting of the directory list box with the Drive setting of drive list box.

Listing 15-2 The code attached to the WINEDIT3.FRM form.

```
'$FORM FileDialogBox
DECLARE SUB FindCom_Click ()
DECLARE SUB LoadCom_Click ()
DECLARE SUB ReadOnlyCom_Click ()
COMMON SHARED GlobalFilename AS STRING
'$INCLUDE: 'CONSTANT.BI'
DIM SHARED NumFiles AS INTEGER
DIM SHARED UseFileDialogBox AS INTEGER
' declare the strings that store the ten text blocks
DIM SHARED KeyStr(1 TO 10) AS STRING


SUB AddFileCom_Click ()
    FileDialogBox.SHOW 1
    IF GlobalFilename <> "" THEN
        NumFiles = NumFiles + 1 ' increment the menu count
        LOAD QuickFilename(NumFiles) ' load a new menu command
        QuickFilename(NumFiles).Caption = GlobalFilename
        QuickFilename(NumFiles).Visible = TRUE
    END IF
END SUB

SUB AppendCom_Click ()
    DIM S AS STRING
    IF TextBox.SelText = "" THEN EXIT SUB
    S = Clipboard.GETTEXT()
    S = S + CHR$(13) + CHR$(10) + TextBox.SelText
    Clipboard.SETTEXT S
END SUB

SUB ClearCom_Click ()
    Clipboard.CLEAR
END SUB

SUB CopyCom_Click ()
    IF Screen.ActiveControl.SelText <> "" THEN
        Clipboard.SETTEXT Screen.ActiveControl.SelText
    END IF
END SUB

SUB CutCom_Click ()
    Clipboard.SETTEXT Screen.ActiveControl.SelText
    Screen.ActiveControl.SelText = ""
END SUB

SUB DeleteFileCom_Click ()
    DIM N AS INTEGER: DIM I  AS INTEGER
    N = VAL(INPUTBOX$("Enter number to delete: "))
    IF (N > 0) AND (N <= NumFiles) THEN
        FOR I = N TO NumFiles - 1
            QuickFilename(I).Caption = QuickFilename(I + 1).Caption
        NEXT I
        UNLOAD QuickFilename(NumFiles)
        NumFiles = NumFiles - 1
    ELSE
        MSGBOX "The number is out-of-range", 0, "Input Error"
    END IF
END SUB

SUB DosCom_Click ()
            ' invoke the DOSBAT batch file
        SHELL ("dosbat.bat " + DosBox.Text)
END SUB

SUB FindBox_Change ()
    FindCom.CHECKED = 0
END SUB

SUB FindCom_Click ()
    STATIC CurIndex AS LONG
    DIM Find AS STRING
    ' update the index to the current character
```

```
CurIndex = TextBox.SelStart + 1
' store the text of the Find box in the variable Find
Find = FindBox.Text
' is variable Find empty?
IF Find = "" THEN EXIT SUB ' nothing to find
' locate the index of the next substring Find in the
' text box
IF CaseSenseChk.Value = 1 THEN
    CurIndex = INSTR(CurIndex + 1, TextBox.Text, Find)
ELSE
    CurIndex = INSTR(CurIndex + 1, UCASE$(TextBox.Text), UCASE$(Find))
END IF
' found a match?
IF CurIndex > 0 THEN
      ' Yes! Display the matching text as selected text
      TextBox.SelStart = CurIndex - 1
      TextBox.SelLength = LEN(Find)
      TextBox.SETFOCUS
      FindCom.CHECKED = TRUE
ELSE
      ' No! Clear any selected text
      TextBox.SelStart = 0
      TextBox.SelLength = 0
      FindCom.CHECKED = FALSE
END IF
END SUB

SUB FindFileCom_Click ()
      ' invoke the DOS dir command with the /s
      ' subdirectory option and the /p paginated
      ' output option
      SHELL ("dosbat.bat dir " + FilenameBox.Text + " /s/p")
END SUB

SUB FindLbl_Change ()
      FindCom.CHECKED = FALSE
END SUB

SUB FindSelTextCom_Click ()
      IF TextBox.SelText <> "" THEN
            FindBox.Text = TextBox.SelText
      END IF
      FindCom_Click
END SUB

SUB Form_Load ()
      NumFiles = 0
      WindowState = 2
      UseFileDialogBox = TRUE
      ReadOnlyCom_Click
END SUB

SUB LoadCom_Click ()
      DIM F AS STRING
      DIM L AS STRING
      DIM NL AS STRING * 2
      IF UseFileDialogBox THEN
            FileDialogBox.SHOW 1
            IF GlobalFilename <> "" THEN
                  FilenameBox.Text = GlobalFilename
            ELSE
                  EXIT SUB
            END IF
      END IF
      ' obtain filename from the filename text box
      F = FilenameBox.Text
      ' if filename is an empty string, exit
      IF F = "" THEN EXIT SUB
      NL = CHR$(13) + CHR$(10)
      ' set error-handler
      ON LOCAL ERROR GOTO BadFile
      ' open the file
      OPEN F FOR INPUT AS 1
      F = "" ' clear variable to reuse it
```

Listing 15-2 Continued.

```
    ' loop to read the text lines from the ASCII file
    DO WHILE NOT EOF(1)
        LINE INPUT #1, L
        F = F + L + NL ' append a new line
    LOOP
    TextBox.Text = F ' copy F into text box
    ' close the file
    CLOSE #1
    ' exit procedure
    EXIT SUB
  ' ************ Error-handler **********
BadFile:
    BEEP
    MSGBOX "Cannot open file " + F, 0, "File I/O Error"
    ON LOCAL ERROR GOTO 0
    RESUME EndOfSub
EndOfSub:
END SUB

SUB PasteCom_Click ()
    Screen.ActiveControl.SelText = Clipboard.GETTEXT()
END SUB

SUB QuickFilename_Click (Index AS INTEGER)
    FilenameBox.Text = QuickFilename(Index).Caption
    UseFileDialogBox = FALSE
    LoadCom_Click
    UseFileDialogBox = TRUE
END SUB

SUB QuitCom_Click ()
    END
END SUB

SUB ReadOnlyCom_Click ()
    SaveCom.Enabled = NOT SaveCom.Enabled
    ReadOnlyCom.CHECKED = NOT SaveCom.Enabled
END SUB

SUB SaveCom_Click ()
    IF (FilenameBox.Text <> "") AND (TextBox.Text <> "") THEN
        ON LOCAL ERROR GOTO BadSaveFilename
        OPEN FilenameBox.Text FOR OUTPUT AS #1
        ON LOCAL ERROR GOTO 0
        PRINT #1, TextBox.Text
        CLOSE #1
    END IF
    EXIT SUB

  ' ***************** Error-Handler ***************
BadSaveFilename:
    ON LOCAL ERROR GOTO 0
    MSGBOX "Cannot write to file " + FilenameBox.Text, 0, "File I/O Error"
    RESUME ExitSaveFile
ExitSaveFile:
END SUB

SUB TextBox_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF Shift = SHIFT_MASK THEN
        SELECT CASE KeyCode
            CASE KEY_F1:  KeyStr(1) = TextBox.SelText
            CASE KEY_F2:  KeyStr(2) = TextBox.SelText
            CASE KEY_F3:  KeyStr(3) = TextBox.SelText
            CASE KEY_F4:  KeyStr(4) = TextBox.SelText
            CASE KEY_F5:  KeyStr(5) = TextBox.SelText
            CASE KEY_F6:  KeyStr(6) = TextBox.SelText
            CASE KEY_F7:  KeyStr(7) = TextBox.SelText
            CASE KEY_F8:  KeyStr(8) = TextBox.SelText
            CASE KEY_F9:  KeyStr(9) = TextBox.SelText
            CASE KEY_F10:  KeyStr(10) = TextBox.SelText
        END SELECT
    ELSEIF Shift = CTRL_MASK THEN
```

```
        SELECT CASE KeyCode
            CASE KEY_F1:  TextBox.SelText = KeyStr(1)
            CASE KEY_F2:  TextBox.SelText = KeyStr(2)
            CASE KEY_F3:  TextBox.SelText = KeyStr(3)
            CASE KEY_F4:  TextBox.SelText = KeyStr(4)
            CASE KEY_F5:  TextBox.SelText = KeyStr(5)
            CASE KEY_F6:  TextBox.SelText = KeyStr(6)
            CASE KEY_F7:  TextBox.SelText = KeyStr(7)
            CASE KEY_F8:  TextBox.SelText = KeyStr(8)
            CASE KEY_F9:  TextBox.SelText = KeyStr(9)
            CASE KEY_F10:  TextBox.SelText = KeyStr(10)
        END SELECT
    END IF
END SUB

SUB ToggleLoadCom_Click ()
    LoadCom.Enabled = NOT LoadCom.Enabled
    ToggleLoadCom.CHECKED = NOT LoadCom.Enabled
END SUB

SUB ToggleOSCom_Click ()
      ' toggle the Visible property of the DosBox text box,
      ' the DosBtn button, the WinBtn button, and the DosLbl
      ' label
    DosBox.Visible = NOT DosBox.Visible
    DosCom.Visible = NOT DosCom.Visible
    DosLbl.Visible = NOT DosLbl.Visible
END SUB
SUB ViewCom_Click ()
    STATIC ViewClipboard AS INTEGER
    STATIC theFilename AS STRING
    STATIC theFileText AS STRING
    IF ViewClipboard THEN
        ViewClipboard = FALSE
        ViewCom.Caption = "View Clipboard"
        ClearCom.Enabled = TRUE
        AppendCom.Enabled = TRUE
        FilenameBox.Text = theFilename
        Clipboard.SETTEXT TextBox.Text
        TextBox.Text = theFileText
        theFileText = ""
    ELSE
        ViewClipboard = TRUE
        ViewCom.Caption = "View File"
        ClearCom.Enabled = FALSE
        AppendCom.Enabled = FALSE
        theFilename = FilenameBox.Text
        theFileText = TextBox.Text
        FilenameBox.Text = "Clipboard"
        TextBox.Text = Clipboard.GETTEXT()
    END IF
END SUB
```

The TreeDir_Change procedure Revises the Filename setting of the file list box with the Path setting of the directory list box.

The FilenameBox_LostFocus Updates the Filename setting of the file list box with the Text setting of the filename text box.

I made the file dialog box form unload itself when finished. This measure preserves the memory used by secondary forms that act as dialog boxes. The price to pay for this feature is the use of the global variable GlobalFilename. You can modify the file dialog box form so that it hides itself when finished. This enables you to store the selected file in an invisible text box that you must also add (call it SelectedFileBox). The primary form is able to obtain the name of the selected file by using the expression FormDialogBox.SelectedFileBox.Text.

Listing 15-3 The code attached to the FILEDLG.FRM form.

```
DECLARE SUB OKBtn_Click ()
COMMON SHARED GlobalFilename AS STRING
'$INCLUDE: 'CONSTANT.BI'


SUB CancelBtn_Click ()
    GlobalFilename = ""
    UNLOAD FileDialogBox
END SUB

SUB DiskDrv_Change ()
    TreeDir.Path = DiskDrv.Drive
END SUB

SUB FileFlb_DblClick ()
    OKBtn_Click
END SUB

SUB FilenameBox_LostFocus ()
      IF FilenameBox.Text = "" THEN EXIT SUB
      FileFlb.Path = FilenameBox.Text
END SUB

SUB Form_Load ()
    FileFlb.Archive = TRUE
    FileFlb.NORMAL = TRUE
    FileFlb.ReadOnly = FALSE
    FileFlb.Hidden = FALSE
    FileFlb.System = FALSE
    FileFlb.Pattern = "*.*"
    FilenameBox.Text = "*.*"
END SUB

SUB OKBtn_Click ()
    IF RIGHT$(TreeDir.Path, 1) <> "\" THEN
          GlobalFilename = TreeDir.Path + "\" + FileFlb.Filename
    ELSE
          GlobalFilename = TreeDir.Path + FileFlb.Filename
    END IF
    UNLOAD FileDialogBox
END SUB

SUB TreeDir_Change ()
    FileFlb.Path = TreeDir.Path
END SUB
```

16
File I/O

The basic file I/O system of Visual Basic greatly resembles that of QuickBasic versions 4 and 4.5. If you are familiar with the file I/O of these implementations you can focus on the examples of this chapter. File I/O is vital in processing information located on disks. Visual Basic also supports the ISAM database structure. The ISAM is a topic beyond the scope of this book. If you plan to use Visual Basic to develop databases, then you should definitely investigate the ISAM features.

The discussions in this chapter assume that you are familiar with file I/O in GW-Basic, BASICA, or QuickBasic. Therefore, the information presented here serves to update you on the Visual Basic file I/O. This assumption also allows me to present the topics of file I/O in a special order. The chapter contains a number of examples that can serve as utilities, including a Visual Basic listing sort utility, a contacts database application, and a text file sorter.

Common file I/O

Before discussing the various types of file I/O, here are a number of functions and statements that work with either type:

The FREEFILE function Automatically returns the next available file handle number. The general syntax for the FREEFILE function is:

```
FileNumber% = FREEFILE
```

The FREEFILE function takes the guesswork out of keeping up with the file handles you have used so far.

The EOF function A Boolean function that returns −1 (TRUE) when you have reached the end of the file. The general syntax for the EOF function is:

```
EOF(fileNumber%)
```

The EOF function enables the program to detect the end of the file when reading a sequence of text lines, records, or single bytes. The EOF function is typically used in a DO-WHILE loop, as shown below:

```
' open the file for input
fileNumber% = FREEFILE
DO WHILE NOT EOF(fileNumber%)
    ' read data
    ' process data
LOOP
```

The LOF function Returns the byte size of an opened file. The general syntax for the LOF function is:

```
LOF(fileNumber%)
```

I recommend assigning the result of the LOF function to a long integer, since file sizes can very well be larger than 32767 bytes (the upper limit of the integer type).

The CLOSE statement Closes one or more files and recuperates their buffers. The general syntax for the CLOSE statement is:

```
CLOSE [#fileNumber1% [,#fileNumber2 [, . . . ]]]
```

You can specify the files to close in a comma-delimited list of file handle numbers. If you want to close all of the files, simply omit the file handle number arguments. This feature also takes the guesswork out of which files to close. Failing to close a file in the output mode might result in the loss of output data (that is, information that is transient in the output buffer).

The KILL statement Allows you to delete a file. The file need not be opened. The general syntax for the KILL statement is:

```
KILL filename$
```

The filename$ is a string expression that can include the drive name, the path, and the wildcard characters ? and *. The latter characters allow you to delete multiple files that fit the specified filename pattern.

The NAME statement Permits you to rename a file and even move a file in a different existing directory. The general syntax for the NAME statement is:

```
NAME oldFileName$ AS newFileName$
```

The oldFileName$ and newFileName$ are string expressions that specify the old and new filenames. These expressions can contain the path of the file. If they specify a different path, the file is moved to the directory specified in the newFileName$ expression. You can move a file to another directory and

maintain the same name—just change the path in the two string expressions. Visual Basic generates errors for the following conditions:

- The file or path specified by oldFileName$ does not exist.
- The paths specified by these string expressions are not on the same disk.
- The path specified in newFileName$ does not exist.
- The file specified by newFileName$ already exists.

☞ **The DIR$ function** Returns the filename that meets the supplied filename pattern. The general syntax for the DIR$ function is:

```
DIR$ [(filePattern$)]
```

The argument filePattern$ usually contains the ? and * wildcard characters to return the names of the matching files. The DIR$ function performs this task sequentially, returning one filename per function call. The rules for using the DIR$ are as follows:

- The first time you call the DIR$ function, supply it with the filePattern$ argument. The function returns the name of the first matching file, or an empty string if no match was found.
- When DIR$ returns the first matching file, you can obtain the next matching file by calling the DIR$ function without an argument. Again, DIR$ returns the name of the next matching file, or an empty string if no match was found.

The DIR$ function is typically used in a DO-WHILE loop, like the one shown below:

```
theFile$ = DIR$ filePattern$
DO WHILE theFile$ <> ''
    ' process theFile$
    theFile$ = DIR$ ' obtain the next matching file
LOOP
```

The above loop enables you to obtain the names of the files matching the specified pattern and process them one at a time. Once the DIR$ function returns an empty string, you should use the filename pattern argument with the next call to DIR$; otherwise, Visual Basic generates an error condition.

Sequential file I/O

Sequential file I/O primarily serves to read and write text files. If you have ever employed file I/O in your Basic programs, it is this type that you most likely used. Using sequential file I/O is compared with reading a scroll—

to get to specific data, you have to read all the information that comes before it.

The various Basic implementations support sequential file I/O to read and write text lines. Each text line is terminated by a pair of carriage return (ASCII 13) and line feed (ASCII 10) characters. These text lines are considered to be variable-length records. The advantages of variable-length records is that they require no extra bytes of storage. The disadvantage is that you cannot instantly tell where the records are located in the file.

Visual Basic offers the following statements to open, close, read, and write from a sequential file:

The OPEN statement Opens a file for input or output. In the case of a sequential file, the general syntax for the OPEN statement is:

```
OPEN filename$ FOR {INPUT ¦ OUTPUT ¦ APPEND} AS [#]fileNumber%
```

The filename$ is a string expression that specifies the filename. You can include the drive and path to access files that are not in the current directory. If the filename$ refers to a nonexistent file, Visual Basic raises an error.

The FOR clause allows you to specify the file mode. The INPUT mode opens an existing file to input. If the file does not exist, Visual Basic raises an error. The OUTPUT and APPEND modes are used to write and append data to a file. If the file is nonexistent, a new file is created (in this case, the APPEND mode works just like the OUTPUT mode).

The AS clause specifies the file handle number used as an index for performing file input and output.

The PRINT# statement Writes data to a sequential file. The general syntax for the PRINT# statement is:

```
PRINT #fileNumber%, expressionList [{; ¦ ,}]
```

Where the fileNumber% parameter is the file handle number. The expressionList contains a comma-delimited list of constants, variables, and expressions that are written to the data file. The expressionList can be followed by a comma or semicolon. The PRINT# statement writes to a file in a manner similar to the way the PRINT statement writes to a form or picture box (and is also the same as the PRINT statement in the other Microsoft Basic implementations). When the expression list is followed by a comma, the next PRINT# output to the same file is tabbed on the same line. When the expression list is followed by a semicolon, the next PRINT# output to the same file appears after the last previous output character.

The WRITE# statement Similar to the PRINT# statement, except that it separates the members of the expression list with commas. The general syntax for the WRITE# statement is:

```
WRITE #fileNumber%, expressionList
```

The INPUT# statement Reads data from a sequential file and stores the data in variables. The general syntax for the INPUT# statement is:

```
INPUT #fileNumber%, variableList
```

The type of data read must match the type of variable in the comma-delimited variableList. Leading spaces on a line are ignored. In the case of numeric variables, the first nonspace character that is encountered is used as the start of a number. Reading this number ends when a space, a comma, or the end of the line is encountered. In the case of strings, leading spaces are also ignored. The first nonspace character is interpreted as the first string character. The input string ends when either a comma or the end of the line is encountered. A blank line yields a 0 when reading a numerical variable, and returns an empty string when reading a string variable.

The LINE INPUT# statement Reads an entire line of text. The general syntax for the LINE inPUT# statement is:

```
LINE INPUT #fileNumber%, stringVariable$
```

The Line Input reads all of the characters up to the carriage return and line feed. Typically, the entire lines of a text file are read using the following kind of DO-WHILE loop:

```
fileNumber% = FREEFILE
OPEN Filename$ FOR INPUT AS #fileNumber%
DO WHILE NOT EOF(fileNumber%)
    LINE INPUT #fileNumber%, TextLine$
    ' process TextLine$
LOOP
CLOSE #fileNumber%
```

The INPUT$ function Returns a specified number of characters from a file. The general syntax for the INPUT$ function is:

```
[stringVariable$ =] INPUT$ (n%, [#]fileNumber%)
```

In the case of sequential files, the INPUT$ function reads the next n% characters from the file whose handle is fileNumber%. These characters include carriage returns, line feeds, and leading spaces; that is, INPUT$ reads every subsequent n% character. The value of n% is in the range of 1 to 32767 for sequential files.

Listing sort utility

Let's look at a Visual Basic program that illustrates sequential file I/O in a way that is relevant to Visual Basic. Visual Basic offers the Save Text option in the File menu to save the code attached to a form or a module in a text file (the default file extension is .TXT). The procedures appear sorted in the output text file.

While Visual Basic for Windows (version 1.0) has a similar feature, the procedures in the output text files appear in the order in which they were typed. This disorder makes it more difficult to locate a particular procedure when you have a long Visual Basic for Windows listing with many procedures.

The next program solves this problem for your friends who use Visual Basic for Windows (and brag about the DOS version.) The program reads the text file produced by Visual Basic for Windows (and also for DOS) and rewrites it such that the procedures are listed alphabetically. The original text file is renamed with a .BAK file extension.

The visual interface of the program contains various types of list boxes, command buttons, and a text box. These controls include the drive, directory, and files list boxes that enable you to locate and select the file you want to process. The command buttons allow you to exit the program, sort the listing file, view a text file, and print a text file. The multiline scrollable text box facilitates viewing text files. I have included the viewing and printing options for the sake of convenience.

When you click on the Sort listing file button, the current selection in the files list box is processed. The program performs the following tasks:

- Reads through the source listing file mainly to count the number of procedures. The heading of the procedures is displayed in the text box in the order in which they appear in the source listing file.
- Reads the source listing file a second time to write into temporary files the lines that are associated with the general declarations section and with each procedure.
- Sorts the data related to procedures using the procedure name as the sort key.
- Builds the target text file by copying the text lines from the temporary files, starting with the temporary file that contains the general declarations section. The other temporary files are in the order of the procedure names. The details of these operations are discussed with the code attached to the program's form.

  Figures 16-1 to 16-3 contain the specifications for the CATALOG.MAK project file and the CATALOG.FRM form file. Figure 16-4 shows a sample session with the program.

Listing 16-1 shows the code attached to the CATALOG.FRM form. Listing 16-1 was obtained by using the Listing Sort Utility program itself. The procedures that are relevant to sorting the Visual Basic procedures are:

The Form_Load procedure Sets the following:

- The WindowState property to 2 (maximized).
- The file pattern to .TXT.

Application Name: Listing Sort Utility	
Application Code Name: CATALOG	
Version: 1.0	Date Created: July 23, 1992
Programmer(s): Namir Clement Shammas	
	List of Filenames
Storage Path:	\VBDOS\EASYVBsa
Project	CATALOG.MAK
Form 1	CATALOG.FRM

16-1 The basic specifications for the CATALOG.MAK project file.

Form # 1	Form Filename: CATALOG.FRM
Version: 1.0	Date: July 23, 1992

Control object type	Default CtlName	Purpose
Drive List Box	Drive1	Shows and selects the current drive.
Directory List	Dir1	Shows and selects the current directory.
File List Box	File1	Shows a group of files.
Text Box	Text1	Specifies the filename pattern.
	Text2	Views and edits a text file.
Command Box	Command1	Exits the program.
	Command2	Sorts the procedures of a listing file.
	Command3	Loads a text file.
	Command4	Saves a text file.
	Command5	Prints a text file.
Label	Label1	Label for filename pattern.
	Label2	Label for drive list box.
	Label3	Label for directory list box.
	Label4	Label for file list box.

16-2 The list of controls in the CATALOG.FRM form file.

- The Archive and Normal file attribute properties to TRUE. The rest of the file attribute properties are set to FALSE.

The GetFileExt function Extracts the file extension from the Filename argument. If the file has a null extension, an empty string is returned. The returned string does not include the dot that separates the primary filename from the file extension.

The GetFilename function Returns the primary filename, including the names of the drive and path if supplied by the Filename argument.

☞ **The GetFunctionName function** Extracts the name of the function

Application (Code) Name: CATALOG
Form # 1
Version: 1.0 Date:

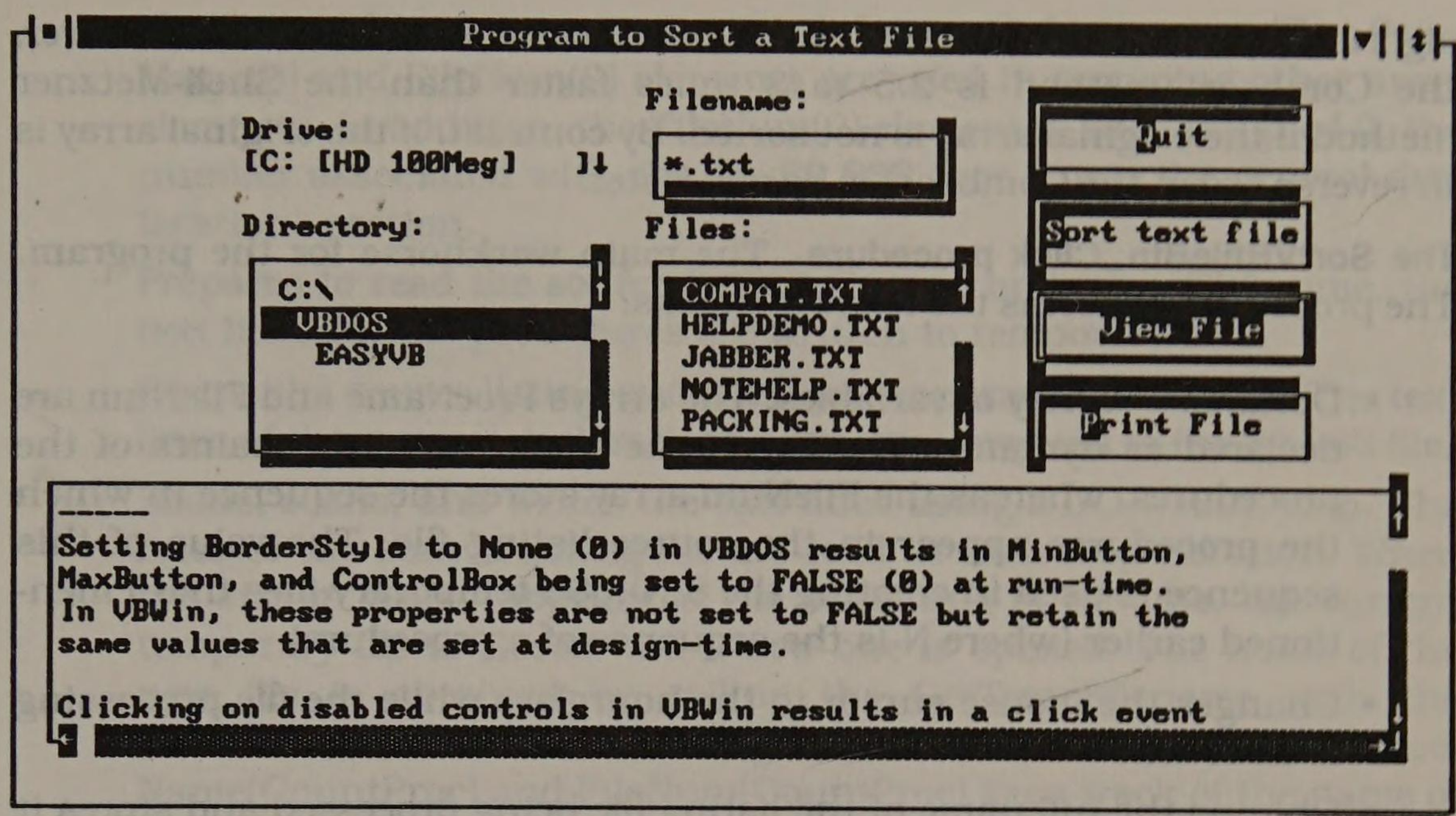
Original control name	Property	New setting
Form	Caption	Program to Sort VB .TXT files.
Drive1	CtlName	DiskDrv
Dir1	CtlName	TreeDir
File1	CtlName	FileFlb
Text1	CtlName	PatternBox
	Text	(empty string)
Text2	CtlName	TextBox
	Text	(empty string)
	MultiLine	TRUE
	ScrollBars	3 - Both
Command1	CtlName	QuitBtn
	Caption	&Quit
	Cancel	TRUE
Command2	CtlName	SortVBfileBtn
	Caption	&Sort VB .TXT file
Command3	CtlName	LoadBtn
	Caption	&Load File
Command4	CtlName	SaveBtn
	Caption	&Save File
Command5	CtlName	PrintBtn
	Caption	&Print File
Label1	CtlName	PatternLbl
	Caption	Filename:
Label2	CtlName	DriveLbl
	Caption	Drive:
Label3	CtlName	DirectoryLbl
	Caption	Directory:
Label4	CtlName	FilesLbl
	Caption	Files:

16-3 The customized settings for the CATALOG.FRM form.

from a text line. The GetFunctionName function assumes that the function declarations appear in the following general format (I am not indenting the format to avoid giving you the impression that leading spaces or tab characters are allowed.):

```
FUNCTION functionName (. . . .
```

The dots mean that what comes after the open parenthesis is not relevant to the GetFunctionName function. The above format is generated by Visual basic and Visual Basic. The difference is in the letter case of the keyword FUNCTION. The Listing Sort Utility assumes that you do not change that



16-4 A sample session with the Listing Sort Utility.

format—the utility is case-insensitive regarding the keyword FUNCTION . This keyword must appear in that form and there must be a single space before and after the function name.

☞ **The GetSubName function** Extracts the name of the Sub procedure from a text line. The GetSubName procedure assumes that the procedure declaration appears in the following general format (I am not indenting the format to avoid giving you the impression that leading spaces or tab characters are allowed.):

SUB procedureName (. . . .

The dots mean that what comes after the open parenthesis is not relevant to the GetFunctionName function. The above format is generated by Visual basic and Visual Basic. The difference is in the letter case of the keyword FUNCTION. The Listing Sort Utility assumes that you do not change that format—the utility is case-insensitive regarding the keyword FUNCTION. This keyword must appear in that form and there must be a single space before and after the function name.

The GetTempFilename function Returns the name of the temporary file using the $⟨N⟩.$$$ filename pattern. The general declarations section is stored in file $0.$$$, the lines of the first procedure are stored in file $1.$$$, and so on.

The function Max Returns the larger value of either argument, I or J.

The SortProcedureData Sorts the arrays that store the names of the procedures and their sequence in the original text file. The SortProcedureData uses the new Combsort method that was developed by Richard Box and Stephen Lacey (see their article in the April 1991 issue of BYTE magazine). The

algorithm of this method resembles that of the Shell-Metzner sort. However, the Combsort method is 2.5 to 3 times faster than the Shell-Metzner method if the original array is not sorted. By contrast, if the original array is in reverse order, the Combsort is hardly faster.

The SortVBfileBtn_Click procedure The main workhorse for the program. The procedure performs the following tasks:

- Declares a variety of variables. The arrays ProcName and FileNum are declared as dynamic. The ProcName array stores the names of the procedures, whereas the FileNum array stores the sequence in which the procedures appear in the source listing file. The value of this sequence is used in creating the $⟨N⟩.$$$ temporary files that I mentioned earlier (where N is the sequence of a procedure).
- Changes the mouse cursor to the hourglass while the file processing is in order.
- Obtains the full name of the listing file being processed and stores it in the variable F.
- Extracts the primary and extension filenames of the input file, and stores them in the variables MainFilename and ExtFilename, respectively.
- Deletes the backup file that has the same primary filename and the .BAK extension name using the KILL statement. Because this file might not exist, use the On Error Resume Next statement to trap the run-time error resulting from trying to delete a nonexistent file. The On Error statement basically tells the program to keep going if the KILL statement raises a run-time error. The error trapping is turned off right after the KILL statement.
- Renames the file extension of the source file listing using the Name statement. This step creates a .BAK backup file out of the original listing file.
- Opens the source file (which was just renamed as a .BAK file) for input.
- Reads the text lines of the source listing file and counts the number of lines that contain the declarations of a Sub or Function procedure. The procedure headings are also posted in the text box.
- Closes the source listing file.
- Examines the value of the CountProc variable. If it is zero, the source file has no procedures. Consequently, the original name of the source listing file is restored, using the Name statement. A message box appears to let you know that the file has no procedures. Finally, the mouse cursor is restored before exiting the procedure.
- Redimensions the arrays ProcName and FileNum using 0 and the variable CountProc as the lower and upper limits, respectively. The 0

elements of both arrays are used for special purpose. The ProcName(0) and FileNum(0) elements are used in swapping other array elements. In addition, the FileNum(0) element is later assigned 0, the number associated with the file $0.$$$ that stores the general declarations section.

- Prepares to read the source listing file a second time. This time, the text lines for the procedures are written to temporary files.
- Opens the source listing and the first temporary file, $0.$$$. The text lines of the general declarations section are written to the $0.$$$ file.
- Reads, scans, and writes the text lines using a DO-WHILE loop. The lines of the source listing file are read in each loop iteration. When the string "SUB " appears at the beginning of the line, the current temporary file is closed and a new one is opened. The name of the new file is obtained by calling the GetTempFilename with the CountProc variable as the argument. The array elements ProcName(CountProc) and FileNum(CountProc) keep track of the name of the scanned procedure and its sequence (which is also its file number).

  When the loop detects the declaration of a function, it performs steps that are similar to those of a Sub procedure.

  The last statement in the DO-WHILE loop writes the currently read text line into the currently opened temporary file.
- Closes the source listing file and the currently opened temporary file.
- Sorts the arrays ProcName and FileNum by calling procedure SortProcedureData. The elements of ProcName are used in ordering both arrays.
- Opens the target listing file for output.
- Uses a FOR-NEXT loop to copy the lines from the temporary files into the target listing file. The FileNum array supplies the arguments for the GetTempFilename function. These arguments specify the name of the temporary files to process in each loop iteration. The FOR-NEXT loop opens, copies, closes, and then deletes the temporary files.
- Closes the target listing file.
- Restores the mouse cursor to its default setting.

Listing 16-2 shows the skeleton code that represents the contents of the Visual basic source listing file that is equivalent to Listing 16-1 (except the Visual Basic for Windows procedures are unordered.) I have shaved off the lines of code from the procedures that have more than two lines. Listing 16-2 shows the order of the procedures as written by Visual Basic for Windows. When you run the Listing Sort utility you can run a text file that contains the procedures in the same order as in Listing 16-1.

Listing 16-1 The code attached to the CATALOG.FRM form.

```
DECLARE FUNCTION GetFilename (Filename AS STRING) AS STRING
DECLARE FUNCTION GetFileExt (Filename AS STRING) AS STRING
DECLARE FUNCTION GetSubName (TextLine AS STRING) AS STRING
DECLARE FUNCTION GetFunctionName (TextLine AS STRING) AS STRING
DECLARE FUNCTION GetTempFilename (N AS INTEGER) AS STRING
DECLARE SUB SortProcedureData (ProcName() AS STRING, FileNum() AS INTEGER, N
AS INTEGER)
DECLARE FUNCTION Max (I AS INTEGER, J AS INTEGER) AS INTEGER
'$INCLUDE: 'CONSTANT.BI'

SUB DiskDrv_Change ()
    ' update file tree path when the disk is changed
    TreeDir.Path = DiskDrv.Drive
END SUB

SUB Form_Load ()
    WindowState = 2
    PatternBox.Text = "*.txt"
    FileFlb.Filename = "*.txt"
    ' set Archive and Normal attributes to True and all
    ' of the other file attributes to False
    FileFlb.Archive = TRUE
    FileFlb.NORMAL = TRUE
    FileFlb.Hidden = FALSE
    FileFlb.ReadOnly = FALSE
    FileFlb.System = FALSE
END SUB

FUNCTION GetFileExt (Filename AS STRING) AS STRING
    ' get the file extension name
    DIM I AS INTEGER
    IF Filename <> "" THEN
        ' locate the "." in the filename
        I = INSTR(Filename, ".")
        ' found the "." ?
        IF I > 0 THEN
            ' get the file extension name
            GetFileExt = MID$(Filename, I + 1, LEN(Filename) - I)
        ELSE
            GetFileExt = "" ' file extension name is a null string
        END IF
    ELSE
        GetFileExt = ""
    END IF
END FUNCTION
FUNCTION GetFilename (Filename AS STRING) AS STRING
    ' get the file name (without the dot delimiter)
    DIM I AS INTEGER
    IF Filename <> "" THEN
        ' locate the "." in the filename
        I = INSTR(Filename, ".")
        ' found the "." ?
        IF I > 0 THEN
            ' extract the filename without the dot delimiter
            GetFilename = LEFT$(Filename, I - 1)
```

```
        ELSE
            GetFilename = Filename ' filename has no dot delimiter
        END IF
    ELSE
        GetFilename = ""
    END IF
END FUNCTION

FUNCTION GetFunctionName (TextLine AS STRING) AS STRING
    ' get the name of the declared Function
    DIM I AS INTEGER: DIM L  AS INTEGER
    ' locate the open parenthesis
    I = INSTR(TextLine, " (") - 1
    L = LEN("FUNCTION ")
    ' extract the name of the function
    GetFunctionName = UCASE$(MID$(TextLine, L + 1, I - L))
END FUNCTION

FUNCTION GetSubName (TextLine AS STRING) AS STRING
    ' get the name of the procedure
    DIM I AS INTEGER: DIM L  AS INTEGER
    ' locate the open parenthesis
    I = INSTR(TextLine, " (") - 1
    L = LEN("SUB ")
    ' extract the file name
    GetSubName = UCASE$(MID$(TextLine, L + 1, I - L))
END FUNCTION

FUNCTION GetTempFilename (N AS INTEGER) AS STRING
    ' return the name of the temporray file #N (as $<N>.$$$)
    GetTempFilename = "$" + FORMAT$(N) + ".$$$"
END FUNCTION

FUNCTION Max (I AS INTEGER, J AS INTEGER) AS INTEGER
    ' return the larger value of I or J
    IF I > J THEN
        Max = I
    ELSE
        Max = J
    END IF
END FUNCTION

SUB PatternBox_KeyPress (KeyAScii AS INTEGER)
    KeyAScii = ASC(LCASE$(CHR$(KeyAScii)))
END SUB

SUB PatternBox_LostFocus ()
    DIM I AS INTEGER
    IF PatternBox.Text = "" THEN PatternBox.Text = "*.*"
    ' locate and remove multiple wildcards
    I = INSTR(PatternBox.Text, " ") ' find first space
    IF I > 0 THEN
        PatternBox.Text = MID$(PatternBox.Text, 1, I - 1)
    END IF
    FileFlb.Filename = PatternBox.Text
END SUB

SUB PrintFileCom_Click ()
```

Listing 16-1 Continued.

```
' performs a simple print task
DIM F AS STRING: DIM TextLine  AS STRING
DIM FileNum AS INTEGER
IF FileFlb.Filename <> "" THEN
    Form1.MousePointer = 11 ' hourglass
    FileNum = FREEFILE ' get the next file buffer
    ' obtain the full filename
    F = TreeDir.Path
    IF RIGHT$(F, 1) <> "\" THEN F = F + "\"
    F = F + FileFlb.Filename
    OPEN F FOR INPUT AS #FileNum ' open file
    ' read each line and print it
    DO WHILE NOT EOF(FileNum)
        LINE INPUT #FileNum, TextLine
        Printer.PRINT TextLine
    LOOP
    Printer.ENDDOC ' close printer buffer
    CLOSE #FileNum ' close file buffer
    Form1.MousePointer = 0 ' restore mouse pointer
  END IF
END SUB

SUB QuitBtn_Click ()
    END
END SUB

SUB SortProcedureData (ProcName() AS STRING, FileNum() AS INTEGER, N AS
INTEGER)
    ' perform a Comb-sort on the arrays ProcName and FileNum.
    ' The arrays are sorted by using the data in ProcName.
    DIM Skip AS INTEGER: DIM InOrder  AS INTEGER
    DIM I AS INTEGER: DIM J  AS INTEGER
     Skip = N%
     DO
         Skip = Max(INT(Skip / 1.3), 1)
         InOrder = TRUE
         FOR I = 1 TO N% - Skip
             J = I + Skip
             IF ProcName$(I) > ProcName$(J) THEN
                    ' swap array members
                 InOrder = FALSE
                 ProcName$(0) = ProcName$(I)
                 ProcName$(I) = ProcName$(J)
                 ProcName$(J) = ProcName$(0)
                 FileNum%(0) = FileNum%(I)
                 FileNum%(I) = FileNum%(J)
                 FileNum%(J) = FileNum%(0)
             END IF
         NEXT I
    LOOP UNTIL InOrder AND (Skip = 1)
END SUB

SUB SortVBfileBtn_Click ()
    DIM NL AS STRING * 2
    DIM F AS STRING
    DIM TextLine AS STRING
```

```
DIM MainFilename AS STRING ' main filename
DIM ExtFilename AS STRING  ' extension name
' declare file I/O channel variables
DIM InputFileNum AS INTEGER
DIM OutputFileNum AS INTEGER
' declare procedure counters
DIM CountProc AS INTEGER: DIM MaxProc  AS INTEGER
' declare dynamic arrays for procedure names
DIM ProcName() AS STRING ' name
DIM FileNum() AS INTEGER ' file number
'-----------------------------------------
IF FileFlb.Filename = "" THEN EXIT SUB
NL = CHR$(13) + CHR$(10)
Form1.MousePointer = 11 ' hourglass
' obtain the full filename
F = TreeDir.Path
IF RIGHT$(F, 1) <> "\" THEN F = F + "\"
F = F + FileFlb.Filename
' extract main filename, including the path
MainFilename = GetFilename(F)
' extract the file extension name
ExtFilename = GetFileExt(F)
' delete previous ".BAK" file
ON LOCAL ERROR RESUME NEXT
KILL MainFilename + ".BAK"
ON LOCAL ERROR GOTO 0  ' turn off error-handler
' rename file as .BAK
NAME F AS MainFilename + ".BAK"
' first pass simply counts the number of
' procedures in the file
InputFileNum = FREEFILE ' get the next file buffer
CountProc = 0 ' initialize procedure counter
TextBox.Text = "" ' clear text box
OPEN MainFilename + ".BAK" FOR INPUT AS #InputFileNum
' loop to read examine each line
DO WHILE NOT EOF(InputFileNum)
    LINE INPUT #InputFileNum, TextLine
    IF (INSTR(UCASE$(TextLine), "SUB ") = 1) OR
       (INSTR(UCASE$(TextLine), "FUNCTION ") = 1) THEN
        CountProc = CountProc + 1
        ' insert procedure heading in the text box
        TextBox.Text = TextBox.Text + TextLine + NL
    END IF
LOOP
CLOSE #InputFileNum ' close file buffer
IF CountProc = 0 THEN ' found no procedures
    ' rename file back to original
    NAME MainFilename + ".BAK" AS F
    MSGBOX "No SUB or FUNCTION were found", 0, "Information"
    Form1.MousePointer = 0 ' restore mouse pointer
    EXIT SUB
END IF
' redimension the arrays
REDIM ProcName(0 TO CountProc) AS STRING
REDIM FileNum(0 TO CountProc)  AS INTEGER
' start pass 2
MaxProc = CountProc
CountProc = 0
```

Listing 16-1 Continued.

```
InputFileNum = FREEFILE ' get the next file buffer
OPEN MainFilename + ".BAK" FOR INPUT AS #InputFileNum
OutputFileNum = FREEFILE ' get the next file buffer
' open the temporary file $0.$$$ that stores the general
' declarations section
OPEN GetTempFilename(CountProc) FOR OUTPUT AS #OutputFileNum
' loop to locate the Sub and Function declarations
DO WHILE NOT EOF(InputFileNum)
    LINE INPUT #InputFileNum, TextLine ' read the next line
    ' does the line contain a Sub procedure?
    IF INSTR(UCASE$(TextLine), "SUB ") = 1 THEN
        CountProc = CountProc + 1
        ' get procedure name
        ProcName(CountProc) = GetSubName(TextLine)
        ' get the name of the next $n.$$$ file
        FileNum(CountProc) = CountProc
        CLOSE #OutputFileNum ' close current file
        OutputFileNum = FREEFILE ' get the next file buffer
          ' open file $n.$$$, where n = CountProc
        OPEN GetTempFilename(CountProc) FOR OUTPUT AS OutputFileNum
    ELSEIF INSTR(UCASE$(TextLine), "FUNCTION ") = 1 THEN
        CountProc = CountProc + 1
        ' get function name
        ProcName(CountProc) = GetFunctionName(TextLine)
        ' get the name of the next $n.$$$ file
        FileNum(CountProc) = CountProc
        CLOSE #OutputFileNum ' close current file
        OutputFileNum = FREEFILE ' get the next file buffer
        ' open file $n.$$$, where n = CountProc
        OPEN GetTempFilename(CountProc) FOR OUTPUT AS OutputFileNum
    END IF
    PRINT #OutputFileNum, TextLine ' write text line to $n.$$$ file
LOOP
' close file buffers
CLOSE #InputFileNum
CLOSE #OutputFileNum
' sort the arrays of procedures
SortProcedureData ProcName(), FileNum(), MaxProc
FileNum(0) = 0 ' set the special value for the element 0
OutputFileNum = FREEFILE ' get the next file buffer
' open the target file
OPEN F FOR OUTPUT AS #OutputFileNum
' get the other $n.$$$ files
FOR CountProc = 0 TO MaxProc
    InputFileNum = FREEFILE ' get the next file buffer
    ' get the $n.$$$ file name
    F = GetTempFilename(FileNum(CountProc))
    OPEN F FOR INPUT AS #InputFileNum
    ' copy the lines of file $n.$$$ into the target file
    DO WHILE NOT EOF(InputFileNum)
        LINE INPUT #InputFileNum, TextLine
        PRINT #OutputFileNum, TextLine
    LOOP
    CLOSE #InputFileNum ' close input buffer
    KILL F ' remove $n.$$$ file
NEXT CountProc
```

```
        CLOSE #OutputFileNum ' close output buffer
        Form1.MousePointer = 0 ' restore mouse cursor
    END SUB

    SUB TextBox_KeyPress (KeyAScii AS INTEGER)
        KeyAScii = 0
    END SUB

    SUB TreeDir_Change ()
        FileFlb.Path = TreeDir.Path
    END SUB
    SUB ViewFileBtn_Click ()
        ' view the target file in TextBox
        DIM F AS STRING: DIM TextLine  AS STRING
        DIM FileNum AS INTEGER
        DIM NL AS STRING * 2
        IF FileFlb.Filename <> "" THEN
            NL = CHR$(13) + CHR$(10)
            FileNum = FREEFILE ' get the next file buffer
            F = TreeDir.Path
            IF RIGHT$(F, 1) <> "\" THEN F = F + "\"
            F = F + FileFlb.Filename
            OPEN F FOR INPUT AS #FileNum
            F = ""
            DO WHILE NOT EOF(FileNum)
                LINE INPUT #FileNum, TextLine
                F = F + TextLine + NL
            LOOP
            CLOSE #FileNum
            TextBox.Text = F
        END IF
    END SUB
```

Listing 16-2 The skeleton code that represents the contents of the original source listing file for Listing 16-1.

```
Sub Form_Load ()
  WindowState = 2
  ...
End Sub

Sub DiskDrv_Change ()
  ' update file tree path when the disk is changed
  TreeDir.Path = DiskDrv.Drive
End Sub

Sub QuitBtn_Click ()
   End
End Sub

Sub TreeDir_Change ()
   FileFlb.Path = TreeDir.Path
End Sub

Sub PatternBox_KeyPress (KeyAScii As Integer)
```

Listing 16-2 Continued.

```
  KeyAScii = Asc(LCase$(Chr$(KeyAScii)))
End Sub

Sub PatternBox_LostFocus ()
  Dim I As Integer
  ...
End Sub

Function GetFilename (Filename As String) As String
' get the file name (without the dot delimiter)
  ...
End Function

Function GetFileExt (Filename As String) As String
' get the file extension name
  ...
End Function

Sub SortVBfileBtn_Click ()
  Dim NL As String * 2
  ...
End Sub

Function GetSubName (TextLine As String) As String
' get the name of the procedure
  ...
End Function
Function GetFunctionName (TextLine As String) As String
' get the name of the declared Function
  ...
End Function

Function GetTempFilename (N As Integer) As String
' return the name of the temporary file #N (as $<N>.$$$)
  GetTempFilename = "$" + Format$(N) + ".$$$"
End Function

Sub SortProcedureData (ProcName$(), FileNum%(), N%)
' perform a Comb-sort on the arrays ProcName and FileNum.
  ...
End Sub

Sub ViewFileBtn_Click ()
' view the target file in TextBox
  ...
End Sub

Sub TextBox_KeyPress (KeyAScii As Integer)
  KeyAScii = 0
End Sub
```

```
Sub PrintFileCom_Click ()
' performs a simple print task
  ...
End Sub

Function Max (I As Integer, J As Integer) As Integer
' return the larger value of I or J
  ...
End Function
```

Random-access file I/O

The data in sequential files are regarded as stored in variable-length records. By contrast, random-access files use fixed-length records to read and write their data. This storage scheme is compared to reading a book—you can open it up to a particular page without scrolling through the previous pages. In fact, you can open random-access files for input, output, or both.

The latter mode is convenient when you need to read, update, and store data in the file. Using fixed-length records, you can zoom in on the record you want to read or write and not be concerned about the other records. Updating random-access files is faster than updating sequential files. The difference in speed increases as the sizes of the two file types increase.

Visual Basic provides the following statements to support random-access file I/O:

The OPEN statement Opens the file for random access using the following general syntax:

OPEN fileName$ FOR RANDOM AS [#]fileNumber% [LEN = recLen%]

The filename$ expression specifies the name of the file to open for random-access. The FOR clause specifies one file mode for random-access file, which allows you to read and write records. The fileNumber% parameter specifies the file handle number, while the LEN clause specifies the length of each record in the file. The default and maximum record lengths are 128 and 32767 bytes, respectively. I strongly recommend that you include the LEN clause when you open a random-access file to read or write data.

The GET statement Enables you to retrieve a record from a random-access file. The general syntax for the GET statement is:

GET [#] fileNumber%, [recordNumber&], recordVariable

The fileNumber% specifies the file handle number. The recordNumber& variable indicates which record you want to read. When this parameter is omitted, the next record is read and the file pointer is advanced by one record. The first record in the file is record # 1. The valid arguments for recordNumber& are 1 to the number of actual records in the file. The run-time system calculates the actual byte position by multiplying the record-

Number& by the record length, recLen%, declared in the Open statement. The largest record number is 2,147,483,647 (which is equal to 2^31 − 1).

The PUT statement Allows you to store a record in a random-access file. The general syntax for the PUT statement is:

```
PUT [#] fileNumber%, [recordNumber&], recordVariable
```

The fileNumber% specifies the file handle number. The recordNumber& variable indicates which record you want to write to. When this parameter is omitted, the next recordVariable is written to the next record and the file pointer is advanced by one record. The first record in the file is record # 1. The valid arguments for recordNumber& are 1 to the 1 plus the number of actual records in the file. Writing to the latter limit enables you to expand the random-access file.

The SEEK statement Allows you to position the file pointer to a specific record for the next read or write operation. The SEEK statement performs no file I/O. The general syntax for the Seek statement is:

```
SEEK [#]fileNumber%, position&
```

The fileNumber% specifies the file handle number. The position& parameter designates the record number. Seeking one record beyond the current number of records expands the random-access file.

The SEEK function Returns a long integer that represents the file pointer of the current record number. The general syntax for the SEEK function is:

```
SEEK(fileNumber%)
```

The fileNumber% specifies the file handle number.

The contacts database application

Random-access files are typically used to maintain the records of a database. The ability to read, write, and update a particular record makes random-access very suitable for the task of maintaining records. The next program implements a database for your contacts. The database stores the person's name, company name, address, city, state, zip code, phone number, and fax number. Figure 16-6 shows the program's form at design time. The figure shows the text boxes and labels that store the above fields for each record. The figure also shows a number of picture boxes, command buttons, and other controls that perform the following tasks:

- The picture boxes with the ⟨⟨ and ⟩⟩ characters to symbolize the left and right arrows. These controls allow you to scroll through your database. When you reach either end of the database, you view the record at the other end. Thus, the picture boxes provide you with a circular list effect.

- The Quit button closes the random-access file and exits the application.
- The Add button inserts a new record into your database.
- The Sort button sorts the records of the database by zip code.
- The Find First button allows you to search for the first record that contains a specified name. You specify the name in an input dialog box.
- The Find Next button allows you to search for the next record that contains a name that you specified earlier.
- The Goto button permits you to move to a specific record by number. Keep in mind that sorting the records changes the record numbers.
- A Delete Record check box that empowers you to mark (and unmark) the record for deletion. The records marked for deletion are actually removed from the database when you click the Quit button.

Figures 16-5 to 16-7 show the specifications for the VBCARD.MAK project file and the VBCARD.FRM form file.

Listing 16-3 contains the declarations of the included file RANDOM.BI. In addition to the TRUE and FALSE constants, this file declares a family of constants that specify the maximum string sizes for the fields of the CardType data type. In addition, these same constants are used to set the limit of how much you can type in the various text boxes.

The CardType declares the fields for the various parts of the contacts record. In addition, a single-character string field IsDel keeps track of whether or not the record is marked for deletion.

Listing 16-4 shows the code attached to the VBCARD.FRM form, which includes the following procedures:

The WriteCurRec Writes the current record (stored in the CurRec variable) to the current record index (stored in the variable CurIndex). The procedure uses a Put statement to write the record at the intended record number.

The ReadCurRec Reads the record indicated by the CurIndex variable into the current record variable, CurRec. The procedure uses a single Get statement to seek and read the target record.

The SaveCurRec Saves the Text property of the various text boxes into the corresponding fields of the current record variable, CurRec. The single character string field, IsDel, stores an X if the delete record check box is checked; otherwise, the IsDel field stores a space character.

The LoadCurRec procedure Copies the data from the CurRec fields into their respective text boxes. The RTrim$ function is used to remove the trailing spaces that exist in the fields of CurRec. The value of the IsDel field is translated into the proper DelRecChk.Value setting.

Application Name: Contacts Database Application
Application Code Name: VBCARD
Version: 1.0 Date Created: July 23, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	VBCARD.MAK
Form 1	VBCARD.FRM

16-5 The basic specifications for the VBCARD.MAK project file.

Form # 1 Form Filename: VBCARD.FRM
Version: 1.0 Date: July 23, 1992

Control object type	Default CtlName	Purpose
Text Box	Text1	Stores the person's name.
	Text2	Stores the company name.
	Text3	Stores the address.
	Text4	Stores the city.
	Text5	Stores the state.
	Text6	Stores the zip code.
	Text7	Stores the phone number.
	Text8	Stores the fax number.
	Text9	Displays the current record number and the total number of records.
Label	Label1	Label for the person's name.
	Label2	Label for the company name.
	Label3	Label for the address.
	Label4	Label for the city.
	Label5	Label for the state.
	Label6	Label for the zip code.
	Label7	Label for the phone number.
	Label8	Label for the fax number.
	Label9	Label for current record number.
	Label10	Label for the Command.
Command Button	Command1	Exits the application.
	Command2	Adds a new record.
	Command3	Sorts the database.
	Command4	Finds the first name.
	Command5	Finds the next name.
	Command6	Goes to a record number.
Picture Box	Picture1	Moves to the previous record.
	Picture2	Moves to the next record.
Check Box	Check1	Toggles the delete status of a record.

16-6 The list of controls in the VBCARD.FRM form file.

Application (Code) Name: VBCARD
Form # 1
Version: 1.0 Date:

Original control name	Property	New setting
Form	Caption	Contacts Card
Text1	CtlName	NameBox
	Text	(empty string)
Text2	CtlName	CompanyBox
	Text	(empty string)
Text3	CtlName	AddressBox
	Text	(empty string)
Text4	CtlName	CityBox
	Text	(empty string)
Text5	CtlName	StateBox
	Text	(empty string)
Text6	CtlName	ZipBox
	Text	(empty string)
Text7	CtlName	PhoneBox
	Text	(empty string)
Text8	CtlName	FaxBox
	Text	(empty string)
Text9	CtlName	CurRecBox
	Text	(empty string)
Label1	CtlName	NameLbl
	Caption	Name:
Label2	CtlName	CompanyLbl
	Caption	Company:
Label3	CtlName	AddressLbl
	Caption	Address:
Label4	CtlName	CityLbl
	Caption	City:
Label5	CtlName	StateLbl
	Caption	State:
Label6	CtlName	ZipLbl
	Caption	Zip:
Label7	CtlName	PhoneLbl
	Caption	Phone:
Label8	CtlName	FaxLbl
	Caption	Fax:
Label9	CtlName	RecNumLbl
	Caption	Record #
Label10	CtlName	FindLbl
	Caption	Find Name
Command1	CtlName	QuitBtn
	Caption	&Quit
Command2	CtlName	AddBtn
	Caption	&Add

16-7 The customized settings for the VBCARD.FRM form.

Command3	CtlName	SortBtn
	Caption	&Sort
Command4	CtlName	FindFirstBtn
	Caption	Find &First
Command5	CtlName	FindNextBtn
	Caption	Find &Next
Command6	CtlName	GotoBtn
	Caption	&Goto
Picture1	CtlName	PrevPct
Picture2	CtlName	NextPct
Check1	CtlName	DelRecChk
	Caption	Delete Record

16-7 Continued.

Listing 16-3 The declarations of the RANDOM.BI included file.

```
Global Const TRUE = -1
Global Const FALSE = 0
Global Const NAME_LEN = 30
Global Const COMPANY_LEN = 40
Global Const ADDRESS_LEN = 40
Global Const CITY_LEN = 15
Global Const STATE_LEN = 2
Global Const ZIP_LEN = 10
Global Const PHONE_LEN = 14

Type CardType
  fName As String * NAME_LEN
  fCompany As String * COMPANY_LEN
  fAddress As String * ADDRESS_LEN
  fCity As String * CITY_LEN
  fState As String * STATE_LEN
  fZip As String * ZIP_LEN
  fPhone As String * PHONE_LEN
  fFax As String * PHONE_LEN
  IsDel As String * 1
End Type
```

The Form_Load Performs the required initialization when you start the VBCARD program. This includes the following tasks:

- Maximizes the form window.
- Writes the "⟨⟨" and "⟩⟩" strings in the picture boxes.
- Assigns the size of the CardType record to the variable RecLen.
- Opens the VBCARD.DAT for random-access with the Len parameter set to the value of the variable RecLen.
- Calculates the number of records in the database and assigns the result to the variable NumRecs.

- If the database is not empty (that is, newly created), read and show the first record. Set the Boolean variable RecNotSaved to false, because the record has been read from the file. By contrast, if the number of records is zero, invoke the AddBtn_Click procedure to add a new record—the most logical step.

The SaveThisRec procedure Ensures that a new or modified record is saved in the database before retrieving another record. The procedure examines the Boolean value of variable RecNotSaved. If that value is −1 (TRUE), the current record is written into the database. This is a two-step process that involves copying the data from the text boxes into the fields of CurRec and then writing the CurRec variable to the file. The RecNotSaved variable is set to 0 (FALSE).

The AddBtn_Click procedure Adds a new record using the following steps:

- Calls the SaveThisRec procedure to make sure that the current record is stored in the file if it is a new record or was modified.
- Increments the total number of records stored in the variable NumRecs.
- Makes the new record the current one.
- Clears the contents of the text boxes by invoking the ClearTextBoxes procedure.
- Updates the record counter text box.

The ClearTextBoxes procedure Resets the text boxes that store the contact's data as follows:

- The name, company, address, city, state, and zip text boxes are cleared.
- The phone and fax text boxes are assigned the template string "( ) −".
- The delete record check box is cleared.

The family of ⟨text⟩Box_Change procedures Ensures that their respective text boxes do not contain more characters than are allowed. The extra characters are deleted. These procedures use the XXXX_LEN global constants to control the maximum string size for each text box. In addition, each procedure sets the Boolean variable RecNotSaved to TRUE, signaling that the record has been modified. A typical example is the CompanyBox_Change procedure, shown below:

```
SUB CompanyBox_Change ()
    IF LEN(CompanyBox.Text) > COMPANY_LEN THEN
        BEEP
        CompanyBox.Text = LEFT$(CompanyBox.Text,
    LEN(CompanyBox.Text) - 1)
```

```
        CompanyBox.SelStart = LEN(CompanyBox.Text)
    END IF
    RecNotSaved = TRUE
END SUB
```

The DelRecChk_Click procedure Toggles the Value setting of the delete record check box and keeps track of the number of deleted records. When the x mark appears in the control, the NumDelRecs variable is incremented by one. When the x mark disappears from the control, the NumDelRecs variable is decremented by one. This value change enables the program to keep track of the exact number of deleted records.

The SortBtn_Click procedure Sorts the records of the database by zip code. The procedure applies the Combsort algorithm to the records of the database. This means that the number of disk accesses increases as the number of records increases—the VBCARD program is not a speed demon when it comes to sorting. The Get and Put statements are used to read and write records.

The FindFirstBtn_Click procedure Finds the first name text box that contains the name you specify. This search feature allows you to search when you know either the first name, the last name, or even part of the last name. The procedure performs the following tasks:

- Saves the current record, if needed.
- Prompts you for the search name using an input dialog box.
- Starts searching the database from the first record. The program loads the first record that contains the name you specified. If no match is found, a message box appears to inform you of the failed search, and the original record you were viewing is restored.

The FindNextBtn_Click procedure Attempts to find the next record that contains the name you specified. The code for this procedure is similar to that of FindFirstBtn_Click. The search begins with the record that comes right after the last matching record.

The GotoBtn_Click procedure Enables you to visit a record by specifying its number. The procedure prompts you for the record number. If your input corresponds to an existing record, that record is loaded in the text boxes. Otherwise, a message box appears to inform you that the value you keyed in is out of range.

The PrevPct_Click and NextPct_Click procedures Allow you to move to the previous and next records, respectively. When the current record index reaches either end of the database, the index is assigned the value of the other end. This assignment implements a wraparound (or circular-list) feature.

The CleanUp procedure Performs an important shutdown process that involves removing the records that are marked for deletion. The problem with the random-access files is that you can expand them, but you cannot truncate them. This means that once a record is inserted it maintains its location in the file. There are various schemes to handle deleted records. I chose the following scheme for the contacts database program:

- A record can be marked for deletion. This does not cause the record to be deleted right away.
- A record marked for deletion can be unmarked, restoring its status as a normal record.
- A record that is marked for deletion is treated like a normal record.
- When you exit the application, the marked records are removed. This is performed by copying the unmarked records of the original database file into another file. The original file is renamed as VBCARD.BAK file extension, while the new file becomes VBCARD.DAT.

The CleanUp procedure implements the last feature and performs the following tasks:

- Closes the database file.
- Determines if there are one or more records to delete.
- Deletes any existing VBCARD.BAK. This step involves error trapping, in case there is no VBCARD.BAK file.
- Renames the VBCARD.DAT database file as VBCARD.BAK.
- Opens both VBCARD.BAK and the new database file VBCARD.DAT as random-access file.
- Uses a FOR-NEXT loop to read the records from VBCARD.BAK and write only the unmarked records in VBCARD.DAT.
- Closes both files.

The QuitBtn_Click procedure Exits the application. The procedure first writes the current record (if that record is new or modified) and then invokes the CleanUp procedure to perform the proper shutdown.

Run the VBCARD program and experiment with adding, searching, sorting, deleting, and visiting various records. You can use this utility to keep track of your daily contacts. Figure 16-8 shows a sample session with program VBCARD.

Binary file I/O

Binary files might be regarded as a special case of random-access files, where the record length is one byte. Binary files delegate the privilege of or-

ganizing the data to the programmer. This privilege means that the programmer must be more careful in reading and writing to the file.

On the other hand, the programmer has the advantage of maintaining complex storage schemes. For example, you can have a binary file store different groups of fixed-length records followed by a single group of variable-length records. Figure 16-9 shows a sample scheme for storing data in such a file:

Programming tip Every sequential and random-access file can be opened as a binary file.

Listing 16-4 The code attached to the VBCARD.FRM form.

```
DECLARE SUB ClearTextBoxes ()
DECLARE SUB LoadCurRec ()
DECLARE SUB SaveCurRec ()
DECLARE SUB WriteCurRec ()
DECLARE SUB ReadCurRec ()
DECLARE SUB SaveThisRec ()
DECLARE SUB AddBtn_Click ()
DECLARE FUNCTION Max (I AS INTEGER, J AS INTEGER) AS INTEGER
DECLARE SUB CleanUp ()
'$INCLUDE: 'RANDOM.BI'
DIM SHARED CurIndex AS LONG
DIM SHARED CurRec AS CardType
DIM SHARED NumRecs AS INTEGER
DIM SHARED NumDelRecs AS INTEGER ' number of deleted records
DIM SHARED RecLen AS INTEGER
DIM SHARED FileNum AS INTEGER
DIM SHARED RecNotSaved AS INTEGER ' Boolean
DIM SHARED FindStr AS STRING
DIM SHARED FoundStr AS INTEGER

SUB AddBtn_Click ()
    SaveThisRec
    NumRecs = NumRecs + 1
    CurIndex = NumRecs
    ClearTextBoxes
    CurRecBox.Text = STR$(CurIndex) + " of " + STR$(NumRecs)
END SUB

SUB AddressBox_Change ()
    IF LEN(AddressBox.Text) > ADDRESS_LEN THEN
        BEEP
        AddressBox.Text = LEFT$(AddressBox.Text, LEN(AddressBox.Text) - 1)
        AddressBox.SelStart = LEN(AddressBox.Text)
    END IF
    RecNotSaved = TRUE
END SUB

SUB CityBox_Change ()
    IF LEN(CityBox.Text) > CITY_LEN THEN
        BEEP
        CityBox.Text = LEFT$(CityBox.Text, LEN(CityBox.Text) - 1)
```

```
        CityBox.SelStart = LEN(CityBox.Text)
    END IF
    RecNotSaved = TRUE
END SUB

SUB CleanUp ()
    DIM FileNum2 AS INTEGER
    DIM I AS INTEGER
    CLOSE
    IF NumDelRecs > 0 THEN
        ON LOCAL ERROR RESUME NEXT
        KILL "VBCARD.BAK"
        ON LOCAL ERROR GOTO 0
        NAME "VBCARD.DAT" AS "VBCARD.BAK"
        FileNum = FREEFILE
        OPEN "VBCARD.BAK" FOR RANDOM AS FileNum LEN = LEN(CurRec)
        FileNum2 = FREEFILE
        OPEN "VBCARD.DAT" FOR RANDOM AS FileNum2 LEN = LEN(CurRec)
        FOR I = 1 TO NumRecs
            GET #FileNum, , CurRec
            IF CurRec.IsDel = " " THEN
                PUT #FileNum2, , CurRec
            END IF
        NEXT I
        CLOSE ' close all buffers
    END IF
    END
END SUB

SUB ClearTextBoxes ()
    NameBox.Text = ""
    CompanyBox.Text = ""
    AddressBox.Text = ""
    CityBox.Text = ""
    StateBox.Text = ""
    ZipBox.Text = ""
    PhoneBox.Text = "(    )     -"
    FaxBox.Text = "(    )     -"
    DelRecChk.Value = 0
END SUB

SUB CompanyBox_Change ()
    IF LEN(CompanyBox.Text) > COMPANY_LEN THEN
        BEEP
        CompanyBox.Text = LEFT$(CompanyBox.Text, LEN(CompanyBox.Text) - 1)
        CompanyBox.SelStart = LEN(CompanyBox.Text)
    END IF
    RecNotSaved = TRUE
END SUB

SUB DelRecChk_Click ()
    IF DelRecChk.Value = 1 THEN
        NumDelRecs = NumDelRecs + 1
    ELSE
        NumDelRecs = NumDelRecs - 1
    END IF
END SUB
```

Listing 16-4 Continued.

```
SUB FaxBox_Change ()
    IF LEN(FaxBox.Text) > PHONE_LEN THEN
        BEEP
        FaxBox.Text = LEFT$(FaxBox.Text, LEN(FaxBox.Text) - 1)
        FaxBox.SelStart = LEN(FaxBox.Text)
    END IF
    RecNotSaved = TRUE
END SUB

SUB FindFirstBtn_Click ()
    DIM OrigCurIndex AS LONG
    DIM I AS INTEGER
    SaveThisRec
    FindStr = INPUTBOX$("Partial name to find", "Name Search")
    IF FindStr <> "" THEN
        Form1.MousePointer = 11
        OrigCurIndex = CurIndex
        FoundStr = FALSE
        FOR CurIndex = 1 TO NumRecs
            ReadCurRec
            I = INSTR(CurRec.fName, FindStr)
            IF I > 0 THEN
                FoundStr = TRUE
                EXIT FOR
            END IF
        NEXT CurIndex
        IF FoundStr THEN
            LoadCurRec
            NameBox.SelStart = I - 1
            NameBox.SelLength = LEN(FindStr)
            NameBox.SETFOCUS
        ELSE
            CurIndex = OrigCurIndex
            ReadCurRec
            LoadCurRec
            MSGBOX "Search found no match", 0, "Information"
        END IF
        Form1.MousePointer = 0
    END IF
END SUB

SUB FindNextBtn_Click ()
    DIM OrigCurIndex AS LONG
    DIM I AS INTEGER
    SaveThisRec
    IF FindStr <> "" THEN
        Form1.MousePointer = 11
        OrigCurIndex = CurIndex
        FoundStr = FALSE
        FOR CurIndex = OrigCurIndex + 1 TO NumRecs
            ReadCurRec
            I = INSTR(CurRec.fName, FindStr)
            IF I > 0 THEN
                FoundStr = TRUE
                EXIT FOR
```

```
        END IF
    NEXT CurIndex
    IF FoundStr THEN
        LoadCurRec
        NameBox.SelStart = I - 1
        NameBox.SelLength = LEN(FindStr)
        NameBox.SETFOCUS
    ELSE
        CurIndex = OrigCurIndex
        ReadCurRec
        LoadCurRec
        MSGBOX "Search found no match", 0, "Information"
        END IF
        Form1.MousePointer = 0
    END IF
END SUB

SUB Form_Load ()
    WindowState = 2 ' maximize window
    PrevPct.AutoRedraw = TRUE
    PrevPct.CurrentX = 0
    PrevPct.CurrentY = 0
    PrevPct.PRINT "<<"
    NextPct.AutoRedraw = TRUE
    NextPct.CurrentX = 0
    NextPct.CurrentY = 0
    NextPct.PRINT ">>"
    RecLen = LEN(CurRec)
    FileNum = FREEFILE
    OPEN "VBCARD.DAT" FOR RANDOM AS #FileNum LEN = RecLen
    NumRecs = LOF(FileNum) / RecLen
    IF NumRecs > 0 THEN
        CurIndex = 1
        GET #FileNum, CurIndex, CurRec
        LoadCurRec
        RecNotSaved = FALSE
    ELSE
        AddBtn_Click
    END IF
END SUB

SUB GotoBtn_Click ()
  ' go to a user-specified page
    DIM S AS STRING
    DIM N AS LONG
    SaveThisRec ' save a new or updated record?
  ' prompt for record number
    S = INPUTBOX$("Go to record number", "Go To Record", "1")
    IF S <> "" THEN
        N = VAL(S)
        IF (N > 0) AND (N <= NumRecs) THEN
            CurIndex = N
            ReadCurRec
            LoadCurRec
        ELSE
            MSGBOX "Record number is out-of-range", 0, "Error!"
        END IF
```

Listing 16-4 Continued.

```
    END IF
END SUB

SUB LoadCurRec ()
    NameBox.Text = RTRIM$(CurRec.fName)
    CompanyBox.Text = RTRIM$(CurRec.fCompany)
    AddressBox.Text = RTRIM$(CurRec.fAddress)
    CityBox.Text = RTRIM$(CurRec.fCity)
    StateBox.Text = RTRIM$(CurRec.fState)
    ZipBox.Text = RTRIM$(CurRec.fZip)
    PhoneBox.Text = RTRIM$(CurRec.fPhone)
    FaxBox.Text = RTRIM$(CurRec.fFax)
    IF CurRec.IsDel <> " " THEN
        DelRecChk.Value = 1
    ELSE
        DelRecChk.Value = 0
    END IF
    CurRecBox.Text = STR$(CurIndex) + " of " + STR$(NumRecs)
END SUB

FUNCTION Max (I AS INTEGER, J AS INTEGER) AS INTEGER
    IF I > J THEN
        Max = I
    ELSE
        Max = J
    END IF
END FUNCTION

SUB NameBox_Change ()
    IF LEN(NameBox.Text) > NAME_LEN THEN
        BEEP
        NameBox.Text = LEFT$(NameBox.Text, LEN(NameBox.Text) - 1)
        NameBox.SelStart = LEN(NameBox.Text)
    END IF
    RecNotSaved = TRUE
END SUB

SUB NextPct_Click ()
    IF NumRecs = 0 THEN EXIT SUB
    SaveThisRec ' save pending record
    CurIndex = CurIndex MOD NumRecs + 1
    ReadCurRec
    LoadCurRec
END SUB

SUB PhoneBox_Change ()
    IF LEN(PhoneBox.Text) > PHONE_LEN THEN
        BEEP
        PhoneBox.Text = LEFT$(PhoneBox.Text, LEN(PhoneBox.Text) - 1)
        PhoneBox.SelStart = LEN(PhoneBox.Text)
    END IF
    RecNotSaved = TRUE
END SUB

SUB PrevPct_Click ()
    IF NumRecs = 0 THEN EXIT SUB
```

```
    SaveThisRec ' save pending record
    IF CurIndex > 1 THEN
        CurIndex = CurIndex - 1
    ELSE
        CurIndex = NumRecs
    END IF
    ReadCurRec
    LoadCurRec
END SUB

SUB QuitBtn_Click ()
    SaveThisRec
    CleanUp
END SUB

SUB ReadCurRec ()
    GET #FileNum, CurIndex, CurRec
END SUB

SUB SaveCurRec ()
    CurRec.fName = NameBox.Text
    CurRec.fCompany = CompanyBox.Text
    CurRec.fAddress = AddressBox.Text
    CurRec.fCity = CityBox.Text
    CurRec.fState = StateBox.Text
    CurRec.fZip = ZipBox.Text
    CurRec.fPhone = PhoneBox.Text
    CurRec.fFax = FaxBox.Text
    IF DelRecChk.Value = 1 THEN
        CurRec.IsDel = "X"
    ELSE
        CurRec.IsDel = " "
    END IF
END SUB

SUB SaveThisRec ()
    IF RecNotSaved THEN
        SaveCurRec
        WriteCurRec
        RecNotSaved = FALSE
    END IF
END SUB

SUB SortBtn_Click ()
  ' sort by Zip
    DIM Skip AS INTEGER: DIM I  AS INTEGER: DIM J  AS INTEGER
    DIM InOrder AS INTEGER
    DIM RecJ AS CardType: DIM RecI  AS CardType
    Form1.MousePointer = 11 ' set mouse to hourglass
    SaveThisRec ' save pending record
    Skip = NumFiles
    DO
        Skip = Max(INT(Skip) / 1.3, 1)
        InOrder = TRUE
        FOR I = 1 TO NumRecs - Skip
            J = I + Skip
            GET #FileNum, I, RecI
```

Listing 16-4 Continued.

```
            GET #FileNum, J, RecJ
            IF RecI.fZip > RecJ.fZip THEN
                  ' swap records
                InOrder = FALSE
                PUT #FileNum, J, RecI
                PUT #FileNum, I, RecJ
            END IF
        NEXT I
    LOOP UNTIL InOrder AND (Skip = 1)
    Form1.MousePointer = 0 ' rstore default mouse
      ' show the first record
        CurIndex = 1
        ReadCurRec
        LoadCurRec
    END SUB

    SUB StateBox_Change ()
        IF LEN(StateBox.Text) > STATE_LEN THEN
            BEEP
            StateBox.Text = LEFT$(StateBox.Text, LEN(StateBox.Text) - 1)
            StateBox.SelStart = LEN(StateBox.Text)
        END IF
        RecNotSaved = TRUE
    END SUB

    SUB WriteCurRec ()
        PUT #FileNum, CurIndex, CurRec
    END SUB

    SUB ZipBox_Change ()
        IF LEN(ZipBox.Text) > ZIP_LEN THEN
            BEEP
            ZipBox.Text = LEFT$(ZipBox.Text, LEN(ZipBox.Text) - 1)
            ZipBox.SelStart = LEN(ZipBox.Text)
        END IF
        RecNotSaved = TRUE
    END SUB
```

Visual Basic offers the following statements to perform binary file I/O:

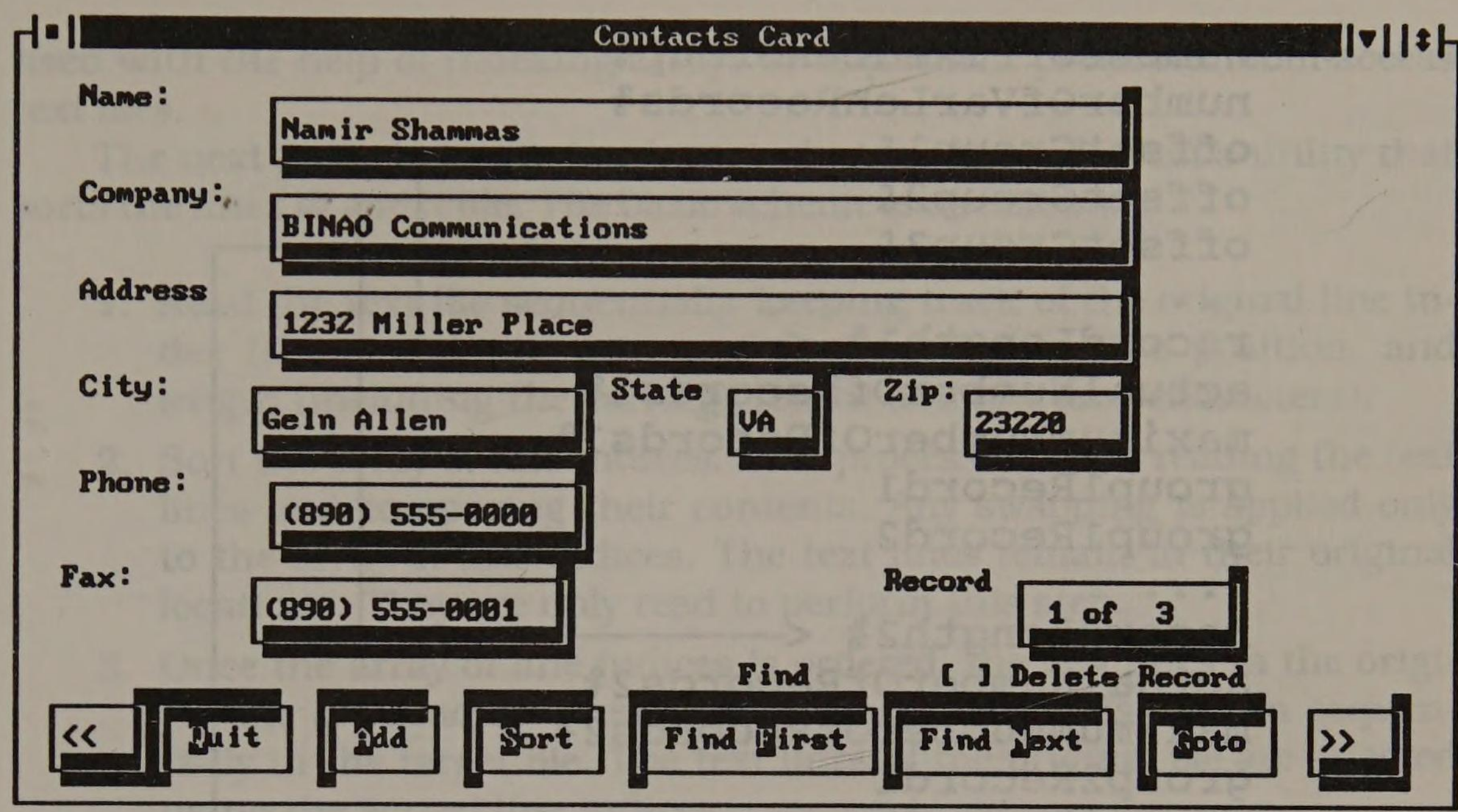
The OPEN statement Opens a file for binary file I/O using the following general form:

OPEN fileName$ FOR BINARY AS [#]FileNumber%

The fileName$ is a string expression that specifies the name of the opened file. The FOR BINARY clause sets the file mode as binary. The AS clause designates the file handle number.

The GET statement Enables you to retrieve one or more bytes from the binary file. The general syntax for the GET statement is:

GET [#] fileNumber%, [byteNumber&], variable



16-8 A sample session with program VBCARD.

The fileNumber% specifies the file handle number. The byteNumber& variable indicates which byte you want to read. When this parameter is omitted, the next LEN(variable) bytes are read. The first byte in the file is byte # 1. The valid arguments for byteNumber& are 1 to the number of actual bytes in the file. The largest byte number is 2,147,483,647 (equal to 2^31 − 1).

When a variable-length string variable is used with the GET statement, the number of bytes read from the binary file equals the current length of the string. This input feature allows you to specify the exact number of bytes to read by using the SPACE$ function. This function is used to create a string with a size equal to the number of bytes to read, as shown below:

```
InputString$ = SPACE$(numberOfBytes%)
GET #fileNumber%,,InputString$
```

The PUT statement Allows you to store one or more bytes in a binary file. The general syntax for the PUT statement is:

```
PUT [#] fileNumber%, [byteNumber&], variable
```

The fileNumber% specifies the file handle number. The byteNumber& variable indicates where you want to write the first byte. When this parameter is omitted, the next LEN(variable) bytes are written. The first byte in the file is byte # 1. When the PUT statement writes beyond the current end-of-file, the file is expanded.

The SEEK statement Allows you to position the file pointer to a specific byte for the next read or write operation. The SEEK statement performs no file I/O. The general syntax for the SEEK statement is:

```
numberOfFixedLenGroup%
numberOfVarLenRecords%
offsetGroup1&
offsetGroup2&
offsetGroup3&
...
recordLength1%
actualNumberOfRecords1%
maximumNumberOfRecords1%
group1Record1
group1Record2
....
recordLength2%
actualNumberOfRecords2%
maximumNumberOfRecords2%
group2Record1
group2Record2
....
recLen1
varLenGroupRec1
recLen2
varLenGroupRec2
recLen3
varLenGroupRec3
...
End-Of-File
```

16-9 A sample scheme for storing groups of fixed-lenth and variable-length records.

SEEK [#]fileNumber%, bytePosition&

The fileNumber% specifies the file handle number. The bytePosition& parameter designates the byte number. Seeking one byte beyond the current number of bytes expands the binary file.

The SEEK function Returns a long integer that represents the file pointer of the current byte number. The general syntax for the SEEK function is:

SEEK(fileNumber%)

The fileNumber% specifies the file handle number.

The text sorting utility

The problem with text files is that they contain variable-length records. The most common text file processing techniques involve sequential input or output. These techniques fail to read a text line more than once without first rereading the text lines before it. The good news is that binary files can be

used with the help of indexing arrays to implement pseudorandom-access text files.

The next program uses the above technique to implement a utility that sorts the lines of a text file. The basic scheme is as follows:

1. Read the text file sequentially, keeping track of the original line index (that is, the sequence of reading it), its byte position, and length (including the carriage return and line feed characters).
2. Sort the array of line indices. This process involves reading the text lines and comparing their contents. Any swapping is applied only to the array of line indices. The text lines remain in their original locations. They are only read to perform this step.
3. Once the array of line indices is ordered, the text lines in the original file are read as pseudorandom-access records written sequentially in the target file. The text lines of the original file are selected using the sorted line indices.

The next program has an interface that is derived from that of program CATALOG, which I presented earlier in this chapter. The only differences are:

- The program form has the caption "Program to Sort Text Files."
- The sort button has the caption "Sort text file" and has the control name of SortFileBtn.

The code attached to the SORTFILE.FRM is shown in Listing 16-5. Most of the procedures in that listing are similar to those in Listing 16-1. The only different procedure is SortFileBtn_Click. This procedure sorts the selected text file using the following tasks:

- Changes the shape of the cursor into the hourglass.
- Obtains the full name of the selected text file.
- Obtains the main and extension names of the selected file.
- Erases any previous .BAK file that has the same primary filename as the selected text file.
- Renames the selected file using the .BAK file extension.
- Opens the selected file as a sequential file.
- Reads the text line in sequence to count the number of lines.
- Closes the selected file.
- If the file is empty, it restores the original filename, displays a message that the file is empty, and exits.
- Dimensions the size of the LineIndex (line index), LinePos (byte position), and LineLen (length of line) arrays using the number of lines.

Listing 16-5 The code attached to the SORTFILE.FRM form.

```
DECLARE FUNCTION GetFilename (Filename AS STRING) AS STRING
DECLARE FUNCTION GetFileExt (Filename AS STRING) AS STRING
DECLARE FUNCTION Max (I AS INTEGER, J AS INTEGER) AS INTEGER
'$INCLUDE: 'CONSTANT.BI'


SUB DiskDrv_Change ()
      ' update file tree path when the disk is changed
    TreeDir.Path = DiskDrv.Drive
END SUB

SUB Form_Load ()
    WindowState = 2
    PatternBox.Text = "*.txt"
    FileFlb.Filename = "*.txt"
    FileFlb.Archive = TRUE
    FileFlb.NORMAL = TRUE
    FileFlb.Hidden = FALSE
    FileFlb.ReadOnly = FALSE
    FileFlb.System = FALSE
END SUB

FUNCTION GetFileExt (Filename AS STRING) AS STRING
    DIM I AS INTEGER
    IF Filename <> "" THEN
        I = INSTR(Filename, ".")
        IF I > 0 THEN
            GetFileExt = MID$(Filename, I + 1, LEN(Filename) - I)
        ELSE
            GetFileExt = ""
        END IF
    ELSE
        GetFileExt = ""
    END IF
END FUNCTION

FUNCTION GetFilename (Filename AS STRING) AS STRING
    DIM I AS INTEGER
    IF Filename <> "" THEN
        I = INSTR(Filename, ".")
        IF I > 0 THEN
            GetFilename = LEFT$(Filename, I - 1)
        ELSE
            GetFilename = Filename
        END IF
    ELSE
        GetFilename = ""
    END IF
END FUNCTION

FUNCTION Max (I AS INTEGER, J AS INTEGER) AS INTEGER
    IF I > J THEN
        Max = I
    ELSE
        Max = J
    END IF
END FUNCTION

SUB PatternBox_KeyPress (KeyAScii AS INTEGER)
    KeyAScii = ASC(LCASE$(CHR$(KeyAScii)))
```

```
END SUB

SUB PatternBox_LostFocus ()
    DIM I AS INTEGER
    IF PatternBox.Text = "" THEN PatternBox.Text = "*.*"
    I = INSTR(PatternBox.Text, " ") ' find first space
    IF I > 0 THEN
        PatternBox.Text = MID$(PatternBox.Text, 1, I - 1)
    END IF
    FileFlb.Filename = PatternBox.Text
END SUB

SUB PrintFileCom_Click ()
    DIM F AS STRING: DIM TextLine  AS STRING
    DIM FileNum AS INTEGER
    IF FileFlb.Filename <> "" THEN
        Form1.MousePointer = 11 ' hourglass
        FileNum = FREEFILE
        F = TreeDir.Path
        IF RIGHT$(F, 1) <> "\" THEN F = F + "\"
        F = F + FileFlb.Filename
        OPEN F FOR INPUT AS #FileNum
        DO WHILE NOT EOF(FileNum)
            LINE INPUT #FileNum, TextLine
            Printer.PRINT TextLine
        LOOP
        Printer.ENDDOC
        CLOSE #FileNum
        Form1.MousePointer = 0
    END IF
END SUB

SUB QuitBtn_Click ()
    END
END SUB

SUB SortFileBtn_Click ()
    DIM LineIndex() AS INTEGER ' index of lines
    DIM LinePos() AS LONG ' byte index for line
    DIM LineLen() AS INTEGER ' size of line
    DIM NumLines AS INTEGER ' number of text lines in file
    DIM I AS INTEGER: DIM J  AS INTEGER   ' indices
    DIM II AS INTEGER: DIM JJ  AS INTEGER ' indices
    DIM Skip AS INTEGER ' used in sorting
    DIM InOrder AS INTEGER ' in-order flag
    DIM LineI AS STRING: DIM LineJ  AS STRING ' lines of text
    DIM F AS STRING ' full filename
    DIM MainFilename AS STRING ' main filename
    DIM ExtFilename AS STRING  ' extension name
      ' declare variables for file I/O channels
    DIM BinFileNum AS INTEGER
    DIM InputFileNum AS INTEGER
    DIM OutputFileNum AS INTEGER
    '-----------------------------------
    IF FileFlb.Filename = "" THEN EXIT SUB
    Form1.MousePointer = 11 ' hourglass
    F = TreeDir.Path
    IF RIGHT$(F, 1) <> "\" THEN F = F + "\"
    F = F + FileFlb.Filename
    MainFilename = GetFilename(F)
    ExtFilename = GetFileExt(F)
      ' delete previous ".BAK" file
```

Listing 16-5 Continued.

```
ON LOCAL ERROR RESUME NEXT
KILL MainFilename + ".BAK"
ON LOCAL ERROR GOTO 0  ' turn off error-handler
  ' rename file as .BAK
NAME F AS MainFilename + ".BAK"
  ' first pass simply counts the number of lines
InputFileNum = FREEFILE
NumLines = 0
OPEN MainFilename + ".BAK" FOR INPUT AS #InputFileNum
DO WHILE NOT EOF(InputFileNum)
    LINE INPUT #InputFileNum, LineI
    NumLines = NumLines + 1
LOOP
CLOSE #InputFileNum
IF NumLines = 0 THEN ' file is empty
      ' rename file back to original
    NAME MainFilename + ".BAK" AS F
    MSGBOX "The text file is empty!", 0, "Information"
    Form1.MousePointer = 0
    EXIT SUB
END IF
  ' redimension the arrays
  ' LineIndex(0) is used in swapping the other elements of LineIndex
REDIM LineIndex(0 TO NumLines) AS INTEGER
REDIM LinePos(1 TO NumLines + 1) AS LONG
    REDIM LineLen(1 TO NumLines) AS INTEGER
    'start pass 2
    LinePos(1) = 1 ' position of first byte
    InputFileNum = FREEFILE
    OPEN MainFilename + ".BAK" FOR INPUT AS #InputFileNum
    LenBigTextLine = 0
    FOR I = 1 TO NumLines
        LINE INPUT #InputFileNum, LineI ' read text line
        LineIndex(I) = I ' set index of line
        LineLen(I) = LEN(LineI) ' store length of line
      ' calculate the byte position of the next line
      ' the 2 represent the Chr$(13)+Chr$(10) characters
      ' that delimit the text lines
    LinePos(I + 1) = LinePos(I) + LineLen(I) + 2
NEXT I
  ' close file buffers
CLOSE #InputFileNum
  ' sort the text lines
BinFileNum = FREEFILE
OPEN MainFilename + ".BAK" FOR BINARY AS #BinFileNum
  ' start the Comb-sort algorithm
Skip = NumLines
DO
    Skip = Max(INT(Skip / 1.3), 1)
    InOrder = TRUE
    FOR I = 1 TO NumLines - Skip
        J = I + Skip
          ' get indirect indices
        II = LineIndex(I)
        JJ = LineIndex(J)
          ' get line I
        LineI = SPACE$(LineLen(II))
```

```
        GET #BinFileNum, LinePos(II), LineI
          ' get line J
        LineJ = SPACE$(LineLen(JJ))
        GET #BinFileNum, LinePos(JJ), LineJ
        IF LineI > LineJ THEN
            InOrder = FALSE
                ' swap LineIndex elements
                ' the element LineIndex(0) is used as a
                ' temporary swap buffer
            LineIndex(0) = LineIndex(I)
            LineIndex(I) = LineIndex(J)
            LineIndex(J) = LineIndex(0)
        END IF
    NEXT I
    LOOP UNTIL InOrder AND (Skip = 1)
      ' now copy lines from .BAK file into the text file
    OutputFileNum = FREEFILE
    OPEN F FOR BINARY AS #OutputFileNum ' open as binary
        FOR I = 1 TO NumLines
            II = LineIndex(I) ' get the I'th line
                ' size string LineI to read the I'th text line
            LineI = SPACE$(LineLen(II) + 2)
                ' use Get and Put to copy lines
            GET #BinFileNum, LinePos(II), LineI
            PUT #OutputFileNum, , LineI
        NEXT I
        CLOSE ' close all buffers
        Form1.MousePointer = 0
    END SUB

    SUB TextBox_KeyPress (KeyAScii AS INTEGER)
        KeyAScii = 0
    END SUB

    SUB TreeDir_Change ()
          FileFlb.Path = TreeDir.Path
    END SUB

    SUB ViewFileBtn_Click ()
        DIM F AS STRING: DIM TextLine  AS STRING
        DIM FileNum AS INTEGER
        DIM NL AS STRING * 2
        IF FileFlb.Filename <> "" THEN
            NL = CHR$(13) + CHR$(10)
            FileNum = FREEFILE
            F = TreeDir.Path
            IF RIGHT$(F, 1) <> "\" THEN F = F + "\"
            F = F + FileFlb.Filename
            OPEN F FOR INPUT AS #FileNum
            F = ""
            DO WHILE NOT EOF(FileNum)
                LINE INPUT #FileNum, TextLine
                F = F + TextLine + NL
            LOOP
            CLOSE #FileNum
            TextBox.Text = F
        END IF
    END SUB
```

- Opens the file for a second sequential pass.
- Reads the file lines using a FOR-NEXT loop to obtain the values for the LineIndex, LinePos, and LineLen arrays.
- Closes the selected file.
- Re-opens the selected file as a binary file.
- Applies the Combsort algorithm to sort the LineIndex array. The Get statements read the various text lines into the variables LineI and LineJ. These string variables are compared to determine whether or not the array LineIndex is ordered. Only the elements of LineIndex are swapped. Notice that the function calls SPACE$(LineLen(II)) and SPACE$(LineLen(JJ)) are used to create string LineI and LineJ with the exact size for the subsequent Get statements.

- Opens the output file using the original name of the selected file. The file is opened as a binary file.
- Copies the text lines using a FOR-NEXT loop. The I'th line is specified by the element LineIndex(I). The text lines are sequentially written to the output file.
- Closes the file buffers.
- Restores the mouse cursor to the default shape.

Run the program and sort a short text file. This should give you an indication that the program is very slow with large text files. However, the program does illustrate the use of binary files to implement the read-only pseudorandom access of text files.

17

Providing on-line help

Chapter 2 discussed the Visual Basic on-line help system. Generally on-line help is becoming more desirable and somewhat expected, especially for non-trivial applications. This chapter covers a method for incorporating on-line help in your own Visual Basic applications.

The process of providing on-line help in your Visual Basic applications involves two steps. The first step uses a help compiler that takes a text file and produces special help files. The second step consists of adding a help form that uses the help files to provide the required on-line help.

The help compiler

The first step in providing on-line help begins with preparing a text file that specifies the help topics and their accompanying text. The accompanying text might include hypertext links; however, this topic is beyond the scope of this book. A help compiler transforms the source file into one or more files that organize the help topics and text into a more suitable form.

The help compiler presented in this chapter is a simple one. It has the following main features and requirements:

- The source file is an ASCII text file.
- The help topics are declared on separate lines (one topic per line) using topic directives that have the following format:

  .Topic topicName

  The line must begin with the topic directive, .Topic. This directive is not case-sensitive, but the topicName is. This means that the current

compiler version allows you to create different topic names that use the same name but have different letter cases. I recommend that you explore this feature with great care. Remember to make the first line a topic directive.

- The lines that are not topic directives make up the text for the associated topic.

The secondary features of the help compiler involve loading, editing, and storing a text file, as well as printing a text file. These features are implemented without "bells and whistles." Feel free to use other utilities to edit and print your text files.

The help compiler uses the constants, user-type, and shared variables. The latter are shown in Listing 17-1. These are identifiers declared in the included file HELP.BI.

The help compiler produces two files. The first file, a .key file, is a random-access file that stores a user-defined type. The second file is a .hlp file that stores the text for the topic in a binary file.

The variables HelpTopic, KeyFilename, and HelpFilename are used to transfer data between the application and the form that provides on-line help.

Figures 17-1 through 17-3 show the specifications for the HELPCOMP.MAK project file and the HELPCOMP.FRM form file. Figure 17-4 shows a sample session with the help compiler.

Listing 17-2 shows the code attached to the HELPCOMP.FRM form. The general declarations section declares the following constants, data type, and variables:

```
' declare the array of topic records
DIM SHARED Topics() AS HelpRec
DIM SHARED F AS STRING ' edited filename
```

The TOPIC_SIZE constant Specifies the length of the fixed string used in the Topic field in the HelpRec, shown next.

The HelpRec user-type Declare three fields, namely, Topic, Offset, and TextLen. The Topic field stores the topic name in a fixed string whose length is specified by the TOPIC_SIZE. If you plan to change the value of TOPIC_SIZE, please do so *before* you use the help compiler to generate help files. The Offset field of the HelpRec indicates the location of the first byte for the text associated with the topic stored in the Topic field. The TextLen field specifies the length of the text, including the carriage return and line field characters.

The MAX_TOPICS constant Specifies the initial upper bound of the array Topics.

The SIZE_INCREMENT constant Defines the increase in the size of array Topics.

Listing 17-1 The lines of the file HELP.BI that contain shared variables.

```
COMMON SHARED HelpTopic AS STRING
COMMON SHARED HelpFilename AS STRING
COMMON SHARED KeyFilename AS STRING
```

Application Name: Help Compiler
Application Code Name: HELPCOMP
Version: 1.0 Date Created: July 24, 1992
Programmer(s): Namir Clement Shammas

	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	HELPCOMP.MAK
Form 1	HELPCOMP.FRM

17-1 The basic specifications for the HELPCOMP.MAK project file.

Form # 1 Form Filename: HELPCOMP.FRM

Version: 1.0 Date: July 24, 1992

Control object type	Default CtlName	Purpose
Drive List Box	Drive1	Shows and selects the current drive.
Directory List	Dir1	Shows and selects the current directory.
File List Box	File1	Shows a group of files.
Text Box	Text1	Specifies the filename pattern.
	Text2	Views and edits a text file.
Command Box	Command1	Exits the program.
	Command2	Compiles a text file.
	Command3	Loads a text file.
	Command4	Saves a text file.
	Command5	Prints a text file.
Label	Label1	Label for filename pattern.
	Label2	Label for drive list box.
	Label3	Label for directory list box.
	Label4	Label for file list box.

17-2 The list of controls in the HELPCOMP.FRM form file.

The HelpRec-typed array Topics This is a dynamic array whose actual size is declared in the CompileBtn_Click procedure and expanded in the procedure ExpandTopicsArray.

The variable F Stores the name of the help source file. I am using a short name to reduce the size of the Visual Basic code lines.

Application (Code) Name: HELPCOMP
Form # 1
Version: 1.0 Date: July 24, 1992

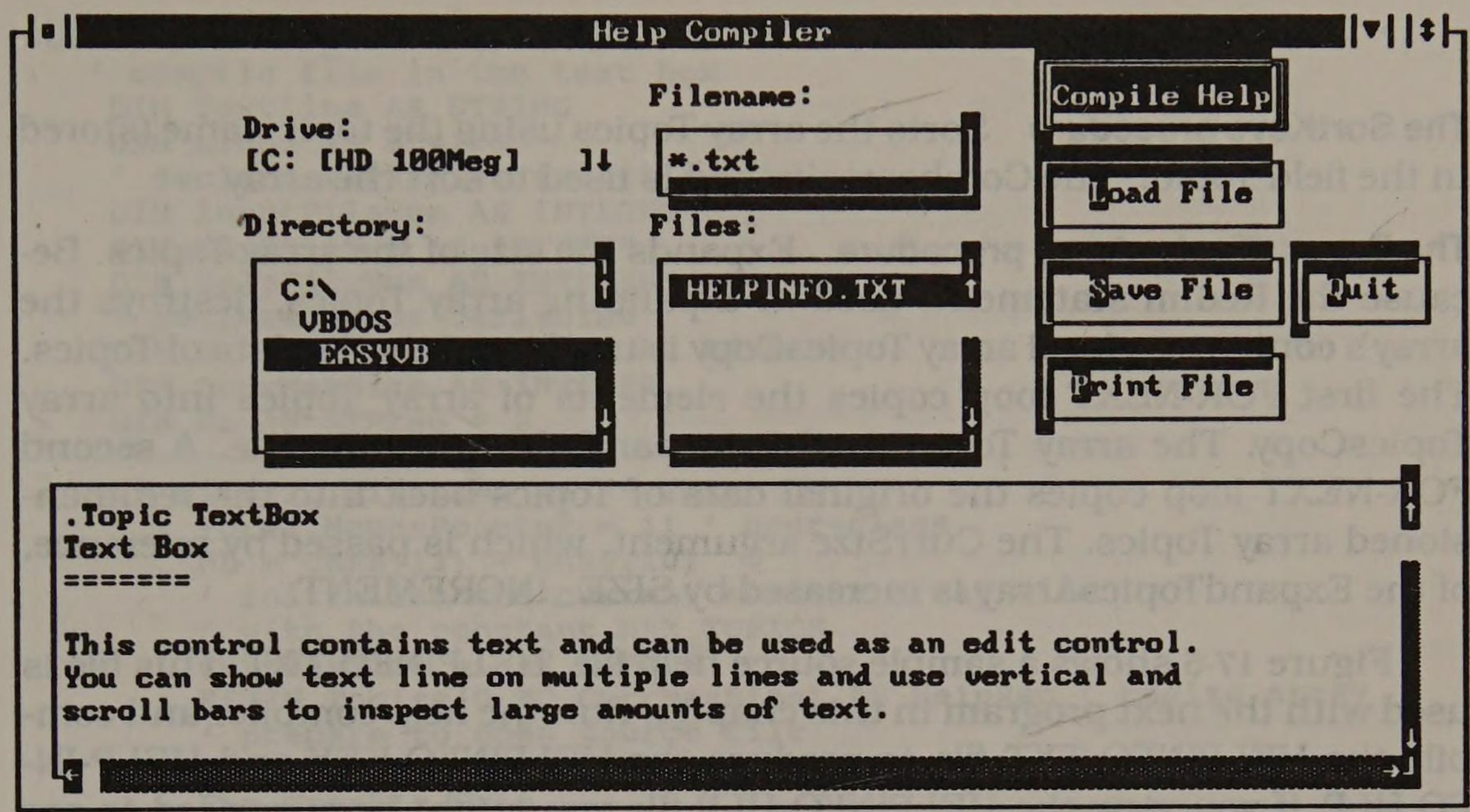
Original control name	Property	New setting
Form	Caption	Help Compiler
Drive1	CtlName	DiskDrv
Dir1	CtlName	TreeDir
File1	CtlName	FileFlb
Text1	CtlName	PatternBox
	Text	(empty string)
Text2	CtlName	TextBox
	Text	(empty string)
	MultiLine	True
	ScrollBars	3 - Both
Command1	CtlName	QuitBtn
	Caption	&Quit
	Cancel	True
Command2	CtlName	CompileBtn
	Caption	&Compile Help File
Command3	CtlName	LoadBtn
	Caption	&Load File
Command4	CtlName	SaveBtn
	Caption	&Save File
Command5	CtlName	PrintBtn
	Caption	&Print File
Label1	CtlName	PatternLbl
	Caption	Filename:
Label2	CtlName	DriveLbl
	Caption	Drive:
Label3	CtlName	DirectoryLbl
	Caption	Directory:
Label4	CtlName	FilesLbl
	Caption	Files:

17-3 The customized settings for the HELPCOMP.FRM form.

The relevant procedures in the HELPCOMP.FRM form are:

The CompileBtn_Click procedure Contains the code that compiles help files and performs the following tasks:

- Ensures that you have selected a file from the files list box. If no selection is made, the procedure does not perform any action.
- Changes the cursor shape to the hour-glass while it is compiling a source help file. The cursor is restored at the end of the procedure.
- Obtains the source filename and opens it as a sequential for input.



17-4 A sample session with the help compiler.

- Obtains the name of the compiled help text file. This file has the same primary name as the input file. The file extension is HLP.
- Opens the help file as a binary file for output.
- Initializes the topic-text variable Lines and the topic counter variable, N.
- Starts a DO-LOOP to process the text lines of the input file. Inside the loop, the procedure reads the text lines and detects the topic directive. When a topic directive is found, the procedure extracts its name and stores in the Topics(N).Topic field. The Offset and TextLine fields are also supplied with data. When the procedure is not processing a topic directive, it accumulates the ordinary text line in the Lines variable. The array Topics is dynamically expanded as needed. If the number of topics equals the value stored in the CurrtMaxSize, the ExpandTopicsArray procedure is called to expand the array by SIZE_INCREMENT elements. When the CompileBtn_Click procedure starts execution, the local variable CurrMaxSize is assigned MAX_TOPICS. Therefore, the constant MAX_TOPICS represents only the initial upper limit.
- When the DO-WHILE loop terminates, the procedure stores the last set of text lines and closes the file buffers.
- Sorts the array Topics using the Topic field. The SortKey procedure is invoked for this purpose.
- Writes the ordered keys to the key file. This file has the same primary name as the input file. The file extension is KEY.

- Closes the key file and restores the mouse cursor.

The SortKeys procedure Sorts the array Topics using the topic name (stored in the field Topic). The Combsort method is used to sort the array.

The ExpandTopicsArray procedure Expands the size of the array Topics. Because the Redim statement, used in expanding array Topics, destroys the array's contents, a local array TopicsCopy is used to retain the data of Topics. The first FOR-NEXT loop copies the elements of array Topics into array TopicsCopy. The array Topics is then expanded to its new size. A second FOR-NEXT loop copies the original data of Topics back into the redimensioned array Topics. The CurrSize argument, which is passed by reference, of the ExpandTopicsArray is increased by SIZE_INCREMENT.

Figure 17-5 shows a sample source help file, HELPINFO.TXT. This file is used with the next program in this chapter. Run the help compiler and compile the HELPINFO.TXT file to produce the HELPINFO.KEY and HELP-INFO.HLP. If you view the HELPINFO.HLP file you might be surprised to see that it looks like the HELPINFO.TXT file, except the topic directives are missing.

The HELPINFO.HLP file is a text file, but a special one. The position and length of the text for each topic is stored in the HELPINFO.KEY file. You might edit this special text file carefully by overwriting characters, but I strongly advise against it.

From a programming perspective, the HELPINFO.HLP file is opened for output as a binary file and not as a sequential file. This enables the CompileBtn_Click procedure to write the text lines accumulated in the variable Lines with a single Print statement.

Listing 17-2 The source code attached to the HELPCOMP.FRM form.

```
DECLARE FUNCTION GetFilename (Filename AS STRING) AS STRING
DECLARE FUNCTION Max (I AS INTEGER, J AS INTEGER) AS INTEGER
DECLARE SUB SortKeys (Topics() AS ANY, N AS INTEGER)
DECLARE SUB ExpandTopicsArray (CurrSize AS INTEGER)

'$INCLUDE: 'CONSTANT.BI'
'$INCLUDE: 'HELP.BI'

CONST TOPIC_SIZE = 40
TYPE HelpRec
    Topic AS STRING * TOPIC_SIZE
    Offset AS LONG
    TextLen AS INTEGER
END TYPE

CONST MAX_TOPICS = 50
CONST SIZE_INCREMENT = 10

' declare the array of topic records
DIM SHARED Topics() AS HelpRec
DIM SHARED F AS STRING ' edited filename
```

```
SUB CompileBtn_Click ()
   ' compile file in the text box
    DIM TextLine AS STRING
    DIM Lines AS STRING
    ' declare variables for file handles
    DIM InputFileNum AS INTEGER
    DIM KeyFileNum AS INTEGER
    DIM HelpFileNum AS INTEGER
    ' declare other variables
    DIM N AS INTEGER: DIM I  AS INTEGER
    DIM CurrMaxSize AS INTEGER
    DIM NL AS STRING * 2
    '----------------------------------------
    IF FileFlb.Filename <> "" THEN
        Form1.MousePointer = 11 ' hour-glass
        NL = CHR$(13) + CHR$(10)
        ' initialize the current maximum array size
        ' with the constant MAX_TOPICS
        CurrMaxSize = MAX_TOPICS
        REDIM Topics(0 TO CurrMaxSize) AS HelpRec ' resize array
        ' prepare to open source file
        InputFileNum = FREEFILE
        F = TreeDir.Path
        IF RIGHT$(F, 1) <> "\" THEN F = F + "\"
        F = F + FileFlb.Filename
        OPEN F FOR INPUT AS #InputFileNum ' open source file
        ' open help file
        HelpFileNum = FREEFILE
        HelpFilename = GetFilename(F) + ".HLP"
        OPEN HelpFilename FOR BINARY ACCESS WRITE AS #HelpFileNum
        Lines = ""
        N = 0 ' initialize number of topics
        ' read lines from source file
        DO WHILE NOT EOF(InputFileNum)
            ' read the next line
            LINE INPUT #InputFileNum, TextLine
            ' does it start with .Topic ?
            IF INSTR(UCASE$(TextLine), ".Topic ") = 1 THEN
                IF Lines <> "" THEN ' write previous help topic text
                    PUT #HelpFileNum, , Lines
                    Topics(N).TextLen = LEN(Lines)
                    Lines = "" ' reset variables
                END IF
                IF N = CurrMaxSize THEN
                    ' resize the array
                    ExpandTopicsArray CurrMaxSize
                END IF
                N = N + 1
                ' obtain the name of the topic
                TextLine = MID$(TextLine, LEN(".TOPIC ") + 1)
                TextLine = LTRIM$(RTRIM$(TextLine))
                Topics(N).Topic = TextLine ' store topic name
                IF N > 1 THEN
                    ' calculate text offset
                    Topics(N).Offset = Topics(N - 1).Offset + _
                                       Topics(N - 1).TextLen
                ELSE
                    Topics(1).Offset = 1
                END IF
            ELSE
                Lines = Lines + TextLine + NL ' accumulates lines
            END IF
        LOOP
        ' store the text for the last topic
```

Listing 17-2 Continued.

```
        PUT #HelpFileNum, , Lines
        Topics(N).TextLen = LEN(Lines) ' obtain length of text
        CLOSE ' close input and .hlp files
        ' now sort the keys
        SortKeys Topics(), N
        ' write the sorted keys to the .KEY file
        KeyFileNum = FREEFILE
        KeyFilename = GetFilename(F) + ".KEY"
        I = LEN(Topics(1))
        OPEN KeyFilename FOR RANDOM ACCESS WRITE AS #KeyFileNum LEN = I
        FOR I = 1 TO N
            PUT #KeyFileNum, , Topics(I)
        NEXT I
        CLOSE ' close all file buffers
        Form1.MousePointer = 0
    END IF
END SUB

SUB DiskDrv_Change ()
    ' update file tree path when the disk is changed
    TreeDir.Path = DiskDrv.Drive
END SUB

SUB ExpandTopicsArray (CurrSize AS INTEGER)
    ' expand the array Topics
    DIM TopicsCopy() AS HelpRec
    DIM I AS INTEGER: DIM NewSize  AS INTEGER
    '-----------------------------------
    REDIM TopicsCopy(0 TO CurrSize) AS HelpRec
    FOR I = 0 TO CurrSize
        TopicsCopy(I) = TopicsCopy(I)
    NEXT I
    NewSize = CurrSize + SIZE_INCREMENT
    REDIM Topics(0 TO NewSize) AS HelpRec
    FOR I = 0 TO CurrSize
        Topics(I) = TopicsCopy(I)
    NEXT I
    CurrSize = NewSize ' update current size
END SUB

SUB Form_Load ()
    WindowState = 2
    PatternBox.Text = "*.txt"
    FileFlb.Filename = "*.txt"
    FileFlb.Archive = TRUE
    FileFlb.NORMAL = TRUE
    FileFlb.Hidden = FALSE
    FileFlb.ReadOnly = FALSE
    FileFlb.System = FALSE
END SUB

FUNCTION GetFilename (Filename AS STRING) AS STRING
    ' get the primary filename, including the drive and path
    DIM I AS INTEGER
    IF Filename <> "" THEN
        I = INSTR(Filename, ".")
        IF I > 0 THEN
            GetFilename = LEFT$(Filename, I - 1)
        ELSE
            GetFilename = Filename
        END IF
    ELSE
```

```
        GetFilename = ""
    END IF
END FUNCTION

SUB LoadFileBtn_Click ()
    ' load text file into the TextBox for viewing and editing
    DIM TextLine AS STRING
    DIM FileNum AS INTEGER
    DIM Lines AS STRING
    DIM NL AS STRING * 2
    '-----------------------------
    IF FileFlb.Filename <> "" THEN
        NL = CHR$(13) + CHR$(10)
        FileNum = FREEFILE
        F = TreeDir.Path
        IF RIGHT$(F, 1) <> "\" THEN F = F + "\"
        F = F + FileFlb.Filename
        OPEN F FOR INPUT AS #FileNum
        Lines = ""
        DO WHILE NOT EOF(FileNum)
            LINE INPUT #FileNum, TextLine
            Lines = Lines + TextLine + NL
        LOOP
        CLOSE #FileNum
        TextBox.Text = Lines
    END IF
END SUB

FUNCTION Max (I AS INTEGER, J AS INTEGER) AS INTEGER
    ' Get the maximum value of either I or J
    IF I > J THEN
        Max = I
    ELSE
        Max = J
    END IF
END FUNCTION

SUB PatternBox_KeyPress (KeyAScii AS INTEGER)
    KeyAScii = ASC(LCASE$(CHR$(KeyAScii)))
END SUB

SUB PatternBox_LostFocus ()
    DIM I AS INTEGER
    IF PatternBox.Text = "" THEN PatternBox.Text = "*.*"
    I = INSTR(PatternBox.Text, " ") ' find first space
    IF I > 0 THEN
        PatternBox.Text = MID$(PatternBox.Text, 1, I - 1)
    END IF
    FileFlb.Filename = PatternBox.Text
END SUB

SUB PrintFileCom_Click ()
    ' simple file printer
    DIM TextLine  AS STRING
    DIM FileNum AS INTEGER
    IF FileFlb.Filename <> "" THEN
        Form1.MousePointer = 11 ' hourglass
        FileNum = FREEFILE
        F = TreeDir.Path
        IF RIGHT$(F, 1) <> "\" THEN F = F + "\"
        F = F + FileFlb.Filename
        OPEN F FOR INPUT AS #FileNum
        DO WHILE NOT EOF(FileNum)
            LINE INPUT #FileNum, TextLine
```

Listing 17-2 Continued.

```
            Printer.PRINT TextLine
        LOOP
        Printer.ENDDOC
        CLOSE #FileNum
        Form1.MousePointer = 0
    END IF
END SUB

SUB QuitBtn_Click ()
    END
END SUB

SUB SaveFileBtn_Click ()
    ' save TextBox in file F
    DIM FileNum AS INTEGER
    IF (F <> "") AND (TextBox.Text <> "") THEN
        FileNum = FREEFILE
        OPEN F FOR OUTPUT AS #FileNum
        PRINT #FileNum, TextBox.Text
        CLOSE #FileNum
    END IF
END SUB

SUB SortKeys (Topics() AS HelpRec, N AS INTEGER)
    ' sort the array Topics using the Topic field.
    ' This procedure uses the new Comb-sort method
    DIM Skip AS INTEGER
    DIM InOrder AS INTEGER
    DIM I AS INTEGER: DIM J  AS INTEGER
    Skip = N
    DO
        Skip = Max(INT(Skip / 1.3), 1)
        InOrder = TRUE
        FOR I = 1 TO N - Skip
            J = I + Skip
            IF Topics(I).Topic > Topics(J).Topic THEN
                InOrder = FALSE
                  ' swap the I and J elements of Topics using
                  ' the element Topic(0)
                Topics(0) = Topics(I)
                Topics(I) = Topics(J)
                Topics(J) = Topics(0)
            END IF
        NEXT I
    LOOP UNTIL InOrder AND (Skip = 1)
END SUB

SUB TreeDir_Change ()
    FileFlb.Path = TreeDir.Path
END SUB
```

Offering on-line help

Once the compiled help files are available, they are ready to be used in the help system that is integrated in your Visual Basic programs. This section includes an example of a simple context-sensitive help system that does an adequate job of providing you with the on-line help you need. The next program is a simple program that shows various mute controls on a form. On-line help is available in one of two ways:

.Topic TextBox
Text Box

This control contains text and can be used as an edit control. You can show text line on multiple lines and use vertical and scroll bars to inspect large amounts of text.

.Topic PictureBox
Picture Box

This control stores pictures and icons.

.Topic VScroller
Vertical Scroller

This control allows you to select from a large number of values.

.Topic HScroller
Horizontal Scroller

This control allows you to select from a large number of values.

.Topic DriveBox
Drive List Box

This control enables you to select new drives.

.Topic DirBox
Directory List Box

This control enables you to select the current directory.

.Topic FileBox
File List Box

This control allows you to select a file.

.Topic OptionButton
Option Button

This control is also known as a radio button. The option buttons are mutually exclusive controls, allowing you to select only one option.

.Topic CheckBox
Check Box

This is a check box. Check boxes allow you to select combinations of options. The check boxes are not mutually exclusive.

.Topic Command
Command Button

This control is perhaps the most popular. By clicking on a command button you generate the Click event that is usually handled by a procedure that animates the command button to perform a task.

.Topic ComboBox
Combo List Box

This control combines the list box and the text box.

.Topic ListBox
List Box

This control presents a list of items to choose from.

17-5 A sample source help file HELPINFO.TXT.

Application Name: Help Demonstration Program	
Application Code Name: HELPDEMO	
Version: 1.0	Date Created:
Programmer(s): Namir Clement Shammas	
	List of Filenames
Storage Path:	\VBDOS\EASYVB
Project	HELPDEMO.MAK
Form 1	HELPDEMO.FRM
Form 2	HELPFORM.FRM

17-6 The basic specifications for the HELPDEMO.MAK project file.

Form # 1	Form Filename: HELPDEMO.FRM
Version: 1.0	Date:

Control object type	Default CtlName	Purpose
TextBox	Text1	Sample text box.
Picture	Picture1	Sample picture box.
Vertical scroll bar	VScroll1	Sample vertical scroll bar.
Horizontal scroll bar	HScroll1	Sample horizontal scroll bar.
Drive List Box	Drive1	Sample drive list box.
Directory List box	Dir1	Sample directory list box.
Files List Box	File1	Sample file list box.
Option Button	Option1	Sample option button.
Check Box	Check1	Sample check box.
Command Button	Command1	Sample command box.
Combo List Box	Combo1	Sample combo list box.
List Box	List1	Sample list box.

17-7 The list of controls in the HELPDEMO.FRM form file.

- Focusing on a control and then pressing the F1 function key. As a result, the program uses a help form to offer help information that is related to the focused control. This is the context-sensitive aspect of the help system. Once you are in the help form, you can select other topics to view their help information.
- Selecting the Help menu to open the help form. From there you can select one or more topics to view their help information.

The help form contains four command buttons, a help-information text box, and a help-topic list box. The command buttons allow you to exit the help form, view the previous help topic, view the next help topic, or view the topic selected in the list box. Double clicking on the topic selected in the list box results in viewing the related help text.

Figures 17-8 through 17-11 show the specifications for the HELP-

Application (Code) Name: HELPDEMO
Form # 1
Version: 1.0 Date:

Original control name	Property	New setting
Form	CtlName	Help Demo Program

17-8 The customized settings for the HELPDEMO.FRM form.

Form # 1 Form Filename: HELPDEMO.FRM
Version: 1.0 Date: July 24, 1992

Caption	Property	Setting	Purpose
&Quit	CtlName	QuitCom	Exits the demo program.
&Help	CtlName	HelpCom	Shows the help form.

17-9, The menu structure for HELPDEMO.FRM.

Form # 2 Form Filename: HELPFORM.FRM
Version: 1.0 Date: July 24, 1992

Control object type	Default CtlName	Purpose
Command Button	Command1	Exits the help form.
	Command2	Selects the previous topic.
	Command3	Selects the next topic.
	Command4	Views the information on the topic selected in List1.
Text Box	Text1	Views the help information.
List Box	List1	Lists the available topics.

17-10 The list of controls in the HELPFORM.FRM form file.

DEMO.MAK project file, HELPDEMO.FRM form file, and the HELPFORM.FRM form file. Figure 17-12 shows the controls on the HELPDEMO.FRM form, while Fig. 17-13 shows a sample session with the help form.

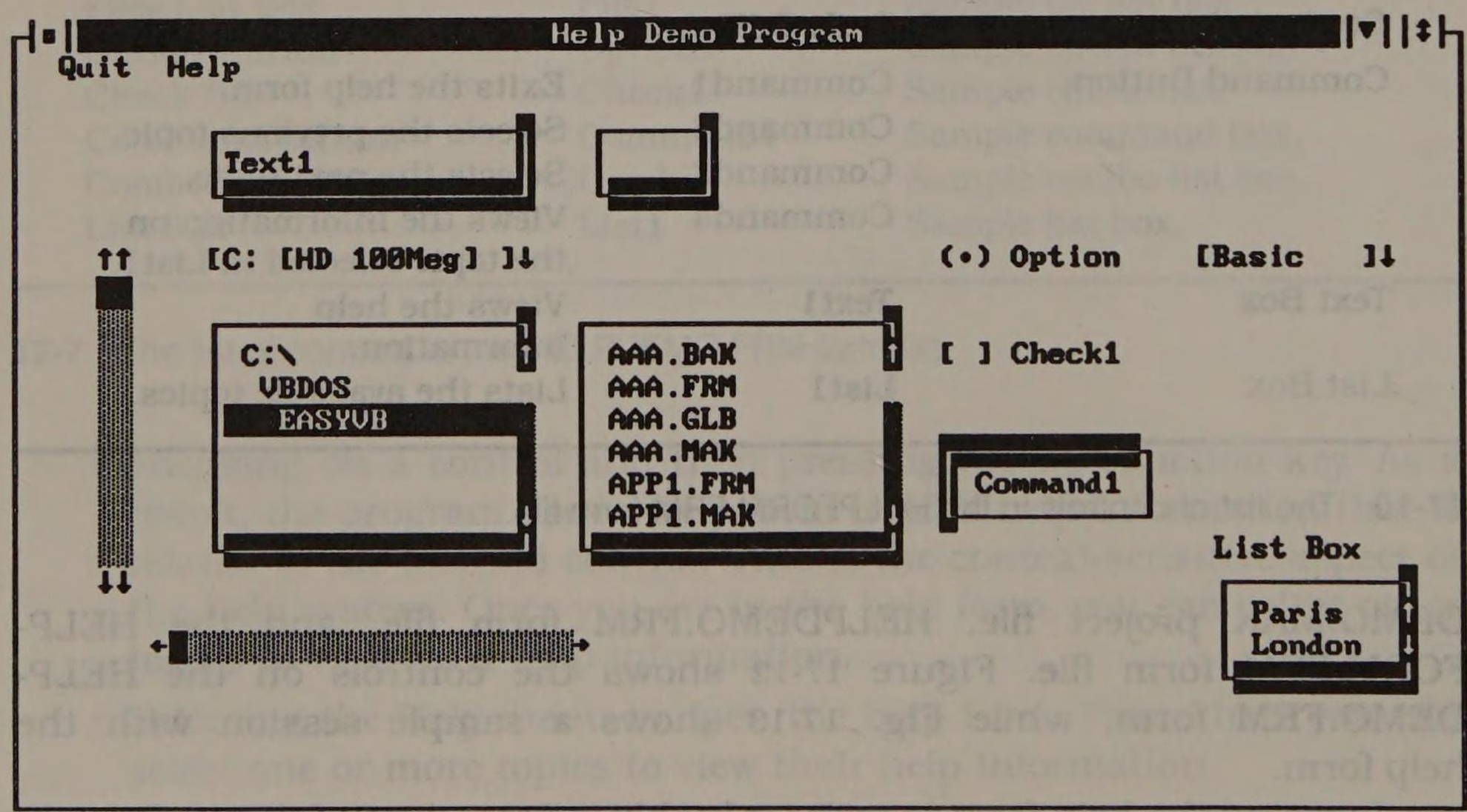
I created the help form to make it highly independent of the application it serves. This enables you to reuse the help form in your Visual Basic applications with little or no recoding.

Listing 17-3 contains the code attached to the HELPDEMO.FRM form. This form communicates with the help form using the global variables HelpTopic, KeyFilename, and HelpFilename. The HelpTopic variable enables the main form (or any other form) to request specific help topic. The help

Application (Code) Name: HELPDEMO
Form # 2
Version: 1.0 Date: July 24, 1992

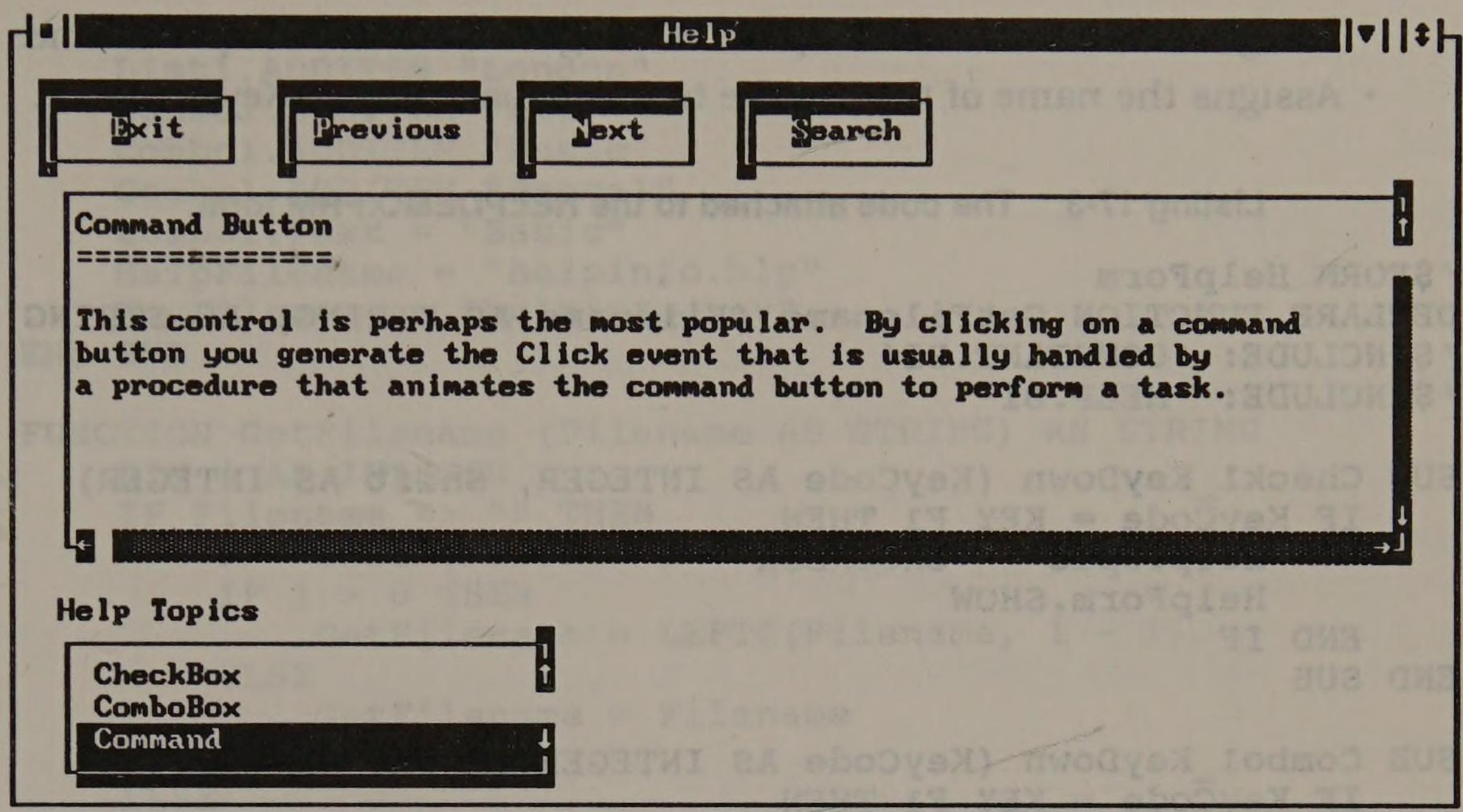
Original control name	Property	New setting
Form	Caption	Help
Command1	CtlName	ExitBtn
	Caption	&Exit
	Cancel	True
Command2	CtlName	PreviousBtn
	Caption	&Previous
Command3	CtlName	NextBtn
	Caption	&Next
Command4	CtlName	SearchBtn
	Caption	&Search
	Default	True
Text1	CtlName	HelpBox
	Text	(empty string)
	MultiLine	True
	ScrollBars	3 - Both
List1	CtlName	TopicsLst

17-11 The customized settings for the HELPFORM.FRM form.



17-12 The controls on the HELPDEMO.FRM form.

form examines the HelpTopic variable to decide which topic to display. The HelpFilename and KeyFilename variables tell the help forms which files to read to provide the on-line help. The relevant procedures in the main form are:



17-13 A sample help session with the HELPFORM.FRM form.

The HelpCom_Click procedure Assigns an empty string to global variable HelpTopic and shows the help form as a modeless window. Because the variable HelpTopic is an empty string, the help form displays an empty help text box.

The XXXX_KeyPress procedures Detect pressing the F1 function key and invoke the help form to display information related to the XXXX control. A sample procedure is shown below:

```
SUB Text1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "TextBox"
        HelpForm.SHOW
    END IF
END SUB
```

The procedure uses an IF-THEN statement to detect pressing the F1 function key. The Then clause has two statements. The first one assigns the name of the text box topic to the global variable HelpTopic. This topic name must also appear in a topic directive, found in the source help file HELPINFO.TXT. The second statement loads and shows the help form. Because no argument is supplied, the help form is modeless; you can select another window before you are finished with the help form.

The Form_Load procedure Performs the following tasks:

- Loads a few data items in the combo list box and in the list box. The purpose of this action is to enhance the visual appearance of these boxes when the program starts running.

- Assigns the name of the help file to the global variable HelpFilename.
- Assigns the name of the Key file to the global variable KeyFilename.

Listing 17-3 The code attached to the HELPDEMO.FRM form.

```
'$FORM HelpForm
DECLARE FUNCTION GetFilename (Filename AS STRING) AS STRING
'$INCLUDE: 'CONSTANT.BI'
'$INCLUDE: 'HELP.BI'

SUB Check1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "CheckBox"
        HelpForm.SHOW
    END IF
END SUB

SUB Combo1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "ComboBox"
        HelpForm.SHOW
    END IF
END SUB

SUB Command1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "Command"
        HelpForm.SHOW
    END IF
END SUB

SUB Dir1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "DirBox"
        HelpForm.SHOW
    END IF
END SUB

SUB Drive1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "DriveBox"
        HelpForm.SHOW
    END IF
END SUB

SUB File1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "FileBox"
        HelpForm.SHOW
    END IF
END SUB

SUB Form_Load ()
    WindowState = 2
```

```
    List1.ADDITEM "Paris"
    List1.ADDITEM "London"
    Combo1.ADDITEM "Ada"
    Combo1.ADDITEM "Basic"
    Combo1.ADDITEM "Pascal"
    Combo1.Text = "Basic"
    HelpFilename = "helpinfo.hlp"
    KeyFilename = "helpinfo.key"
END SUB

FUNCTION GetFilename (Filename AS STRING) AS STRING
    DIM i AS INTEGER
    IF Filename <> "" THEN
        i = INSTR(Filename, ".")
        IF i > 0 THEN
            GetFilename = LEFT$(Filename, i - 1)
        ELSE
            GetFilename = Filename
        END IF
    ELSE
        GetFilename = ""
    END IF
END FUNCTION

SUB HelpCom_Click ()
    HelpTopic = "" ' no particular help topic
    HelpForm.SHOW
END SUB

SUB HScroll1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "HScroller"
        HelpForm.SHOW
    END IF
END SUB

SUB List1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "ListBox"
        HelpForm.SHOW
    END IF
END SUB

SUB Option1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "OptionButton"
        HelpForm.SHOW
    END IF
END SUB

SUB Picture1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "PictureBox"
        HelpForm.SHOW
```

Listing 17-3 Continued.

```
    END IF
END SUB

SUB QuitBtn_Click ()
        END
END SUB

SUB QuitCom_Click ()
    END
END SUB

SUB Text1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "TextBox"
        HelpForm.SHOW
    END IF
END SUB

SUB VScroll1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    IF KeyCode = KEY_F1 THEN
        HelpTopic = "VScroller"
        HelpForm.SHOW
    END IF
END SUB
```

Listing 17-4 shows the code attached to the HELPFORM.FRM form. The listing contains the following variables:

- The NumTopics variable stores the number of topics found in the file KeyFilename. This variable is also used to specify the upper limit of the dynamic array Topics.
- The Topics array is declared as a dynamic array.
- The KeyFileNum variable stores the file handle number for the KeyFilename.
- The HelpFileNum variable stores the file handle number for the HelpFilename.
- The TopicIndex keeps track of the index of the current topic.

The relevant procedures of the help form are as follows:

The ExitBtn_Click procedure Closes all of the file buffers before unloading the help form.

The Form_Load procedure Performs the required initialization when the help form is loaded, achieved by the following set of tasks:

- Sets the WindowState property to 2, maximizing the window of the help form.

- Opens the key and help data files.
- Calculates the number of topics and assigns the result to the variable NumTopics.
- Redimensions the array Topics using NumTopics to specify the upper limit.
- Reads the key file that contains the HelpRec-typed data and stores the information in the Topics array.
- Inserts the topic names in the TopicsLst list box.
- Tests if the global variable HelpTopic is not an empty string. If the tested condition is true, then the main form has requested help information on a topic stored in HelpTopic. The Then clause invokes the BinSearch function to locate the index of the help topic. If the index is greater than zero (this is an extra precaution) then load the text for the requested topic.

The BinSearch function Performs a binary search on the ordered elements of array Topics and returns the index of the matching array element. If no match is found the function returns 0. Notice that the function uses a local fixed-string FindStr in searching for the sought element. This search compares FindStr with the fixed-string fields Topic of the various elements of array Topics. If you replace FindStr with Find (the supplied argument), the BinSearch will most likely fail to locate the specific topic because of the spaces that are padded to the Topic field. Using FindStr gives a faster function execution than if the expression RTRIM$(Topics(Median).Topic) is used.

The LoadTopic procedure Loads the topic specified by the parameter Index. The function checks the validity of the argument for Index before reading the appropriate help text from the help file. The local variable S is used to read the help text in one swoop. Once the text is read the contents of variable S are then assigned to the Text property of the HelpBox text box. The TopicIndex variable is assigned the value of the Index argument.

The TopicsLst_DblClick Loads the topic selected in the TopicsLst list box. The first statement puts the selected topic into the variable HelpTopic. The second statement invokes the BinSearch and the LoadTopic procedure to retrieve the requested help information.

The SearchBtn_Click Simply issues a call to TopicsLst_DblClick to load the help information for the topic selected in the TopicsLst list box.

The PreviousBtn_Click and NextBtn_Click procedures Load the previous and next topics, respectively. Both procedures are coded to wrap around the topics when you reach either the first or last topic.

Run the help demonstration program. Select a control and press the F1 function key. This takes you to the help form and displays the help informa-

Listing 17-4 The code attached to the HELPFORM.FRM form.

```
DECLARE FUNCTION BinSearch (Find AS STRING) AS INTEGER
DECLARE SUB LoadTopic (BYVAL Index AS INTEGER)
DECLARE SUB TopicsLst_DblClick ()

CONST TOPIC_SIZE = 40

TYPE HelpRec
  topic AS STRING * TOPIC_SIZE
  Offset AS LONG
  TextLen AS INTEGER
END TYPE

'$INCLUDE: 'HELP.BAS'

DIM SHARED NumTopics AS INTEGER
DIM SHARED Topics() AS HelpRec
DIM SHARED KyFileNum AS INTEGER
DIM SHARED HelpFileNum AS INTEGER
DIM SHARED TopicIndex AS INTEGER


FUNCTION BinSearch (Find AS STRING) AS INTEGER
      ' performs a binary search on the topic
    DIM Loas AS INTEGER: DIM High  AS INTEGER
    DIM Median AS INTEGER
    DIM FindStr AS STRING * TOPIC_SIZE
    FindStr = Find ' make a String * TOPIC_SIZE version
    Low = 1
    High = NumTopics
    DO
        Median = (Low + High) \ 2
        IF FindStr < Topics(Median).topic THEN
            High = Median - 1
        ELSE
            Low = Median + 1
        END IF
    LOOP UNTIL (FindStr = Topics(Median).topic) OR (Low > High)
    IF FindStr = Topics(Median).topic THEN
        BinSearch = Median
    ELSE
        BinSearch = 0
    END IF
END FUNCTION

SUB ExitBtn_Click ()
    CLOSE ' close all of the help file buffers
    UNLOAD HelpForm
END SUB

SUB Form_Load ()
    DIM Index AS INTEGER
    DIM HelpVar AS HelpRec
    '---------------------------
    WindowState = 2
      ' open .key file
    KeyFileNum = FREEFILE
    OPEN KeyFilename FOR RANDOM ACCESS READ AS #KeyFileNum LEN = LEN(HelpVar)
      ' open .hlp file
    HelpFileNum = FREEFILE
    OPEN HelpFilename FOR BINARY ACCESS READ AS #HelpFileNum
      ' calculate the number of topics
    NumTopics = LOF(KeyFileNum) / LEN(HelpVar)
    REDIM Topics(0 TO NumTopics)  AS HelpRec' custom-size the array
      ' insert topic names in the topic list
    FOR Index = 1 TO NumTopics
          ' read a topic record
        GET #KeyFileNum, , Topics(Index)
          ' add topic name to the list
```

```
        TopicsLst.ADDITEM RTRIM$(Topics(Index).topic)
    NEXT Index
      ' did main menu request help on a particular topic?
    IF HelpTopic <> "" THEN
        Index = BinSearch(HelpTopic) ' get the topic index
        IF Index > 0 THEN LoadTopic (Index)' load topic
    END IF
END SUB

SUB HelpBox_KeyPress (KeyAScii AS INTEGER)
    KeyAScii = 0
END SUB

SUB LoadTopic (BYVAL Index AS INTEGER)
      ' load a topic by index
    DIM S AS STRING
      ' is index within range?
    IF (Index > 0) AND (Index <= NumTopics) THEN
          ' create string buffer equal to the length of the
          ' related topic
        S = SPACE$(Topics(Index).TextLen)
          ' read the topic text
        GET #HelpFileNum, Topics(Index).Offset, S
        HelpBox.Text = S ' assign to help text box
        TopicIndex = Index ' store the topic index
    END IF
END SUB

SUB NextBtn_Click ()
      ' find next topic
    TopicIndex = TopicIndex MOD NumTopics + 1
    LoadTopic (TopicIndex)
END SUB

SUB PreviousBtn_Click ()
      ' find previous topic
    IF TopicIndex > 1 THEN
        TopicIndex = TopicIndex = 1
    ELSE
        TopicIndex = NumTopics
    END IF
    LoadTopic (TopicIndex)
END SUB

SUB SeachBtn_Click ()
    TopicsLst_DblClick
END SUB

SUB TopicsLst_DblClick ()
      ' load topic selected in the list box
    HelpTopic = TopicsLst.Text
    LoadTopic (BinSearch(HelpTopic))
END SUB
```

tion related to the selected control. Click on the Previous and Next buttons to scroll through the various help topics. In addition, select topics at your discretion from the list box and double click to see the related help information. When you are finished experimenting with the help window, click the Exit button. You can select another window when the help window appears. However, you cannot select another help topic from the main form before you close the help form.

A

Visual Basic for DOS language: Quick reference

This appendix contains a quick reference for the Visual Basic programming language. If you are familiar with QuickBasic, then you are familiar with most of the language syntax of Visual Basic. If you are only familiar with GW-BASIC, BASICA, or other BASIC dialects, this reference should give you a brief idea of the syntax of Visual Basic. The material here is not aimed at teaching you about the Visual Basic language. Instead, it is meant to offer you a quick reference and to update you on the basic constructs of Visual Basic.

Data types

Visual Basic supports the six data types shown in Table A-1. The Currency data type is introduced in Visual Basic to maintain a floating-point that is dedicated for financial calculations.

The type names in Table A-1 are reserved keywords used by Visual Basic in declaring variables and user-defined types.

Literal constants

Literal constants present explicit constants in one of the basic data types. Table A-2 offers examples of the literal constants for each basic type.

Visual Basic supports both dynamic (variable-length) and fixed-length strings. Dynamic strings are declared using the String keyword. Fixed-length strings are declared using the following syntax:

```
STRING * size
```

Table A-1 The predefined basic data types.

Type name	Description	Type-declaration character	Range
Integer	%	2-byte integer	−32768 to 32767
Long	&	4-byte integer	−2,147,483,648 to 2,147,483,647
Single (default)	!	4-byte floating-point number	−3.37E+38 to 3.37E+38
Double	#	8-byte floating-point number	−1.67D+308 to 1.67D+308
Currency	@	Number with fixed decimal point	−9.22E+14 to 9.22E+14
String	$	String	N/A

Table A-2 Example of literal constants.

Type name	Example	Comments
INTEGER	−12, 344	
	&O11	octal number
	&HFFFF	hexadecimal number
LONG	100000, −345245	
	&O123&	octal number
	&HFFFFF&	hexadecimal number
SINGLE (default)	−345.2, 1.2E+11, 2.3E−09	
DOUBLE	−345.664, 2.34D+98, 2.3D−09	
CURRENCY	12.33	
STRING	"Hello there"	

The size parameter determines the length of the fixed string. A fixed-length string retains the same size.

Operators

Visual Basic supports the assignment, math, string, relational, Boolean, and access operators, as shown in Table A-3.

Identifiers

Identifiers are the names of data types, constants, variables, operators, procedures, and functions. These names are either predefined (also called reserved words) by Visual Basic or they are user-defined. An identifier must follow these rules:

- It can be no more than 40 characters long.
- It must start with a letter.

Table A-3 The Visual Basic operators.

Operator	Unary?	Class	Purpose	Example
+	Yes	Math	Plus sign	+2
	No	Math	Addition	2 + I%
	No	String	Concatenate	FirstName$ + "Shammas"
−	Yes	Math	Negate	−2 * X#
	No	Math	Subtract	23.4 − Rate!
*	No	Math	Multiply	Length# * Width#
/	No	Math	Floating-point divide	355.0 / 113.0
\	No	Math	Integer division	355 \ 113
MOD	No	Math	Integer modulo	355 MOD 113
^	No	Math	Exponentiation	10 ^ X#
=	No	General	Assignment	X# = 10 ^ 2.3
	No	Relational	Test if equal to	I% = J%
<	No	Relational	Test if less than	I% < J%
>	No	Relational	Test if greater than	I% > J%
>=	No	Relational	Test if greater than or equal to	I% >= J%
<=	No	Relational	Test if less than or equal to	I% <= J%
AND	No	Boolean	Logical/bitwise AND	(I% = 0) AND (J% = 0)
OR	No	Boolean	Logical/bitwise OR	(I% = 0) OR (J% = 0)
XOR	No	Boolean	Logical/bitwise XOR	(I% = 0) XOR (J% = 0)
NOT	Yes	Boolean	Logical/bitwise NOT	NOT ((I% = 1) AND (J% = 1))
IMP	No	Boolean	Implication (first operand is false, or second operand true)	(I% = 0) IMP (J% = 1)
EQV	No	Boolean	Logical equivalence	(I% = 0) EQV (J% = 0)
TYPE-OF-IS	No	Boolean	Test type of control	TYPEOF Btn IS Option
()	Yes	General	Indexed access	X# = A(I%)
.	No	General	Field access	X# = MyRec.Weight

- It can only include digits and the underscore character.
- It cannot be the same as a reserved keyword; however, these keywords can be embedded in an identifier.

Examples of identifiers include:

myName
Your_name
Your_Name
R2D2
Day10
Day_10_Rate

Visual Basic identifiers are not case-sensitive. This means that identifiers like MyName, myName, myname, and MYNAME are treated as the same identifier.

Symbolic constants

Symbolic constants are identifiers that are associated with literal constants. Symbolic constants provide a much clearer program readability and empower you to change the constant value in one location. The general syntax for declaring constants is:

```
CONST constantName1 = value1 [, constantName2 = value2, . . . ]
```

Some examples of declaring constants are shown below:

```
CONST TRUE = -1, FALSE = 0
CONST SecPerMinute = 60.0
    CONST SecPerHour = 3.6D+3
```

To use constants in multiple forms and modules, place their declaration in an included file and use the INCLUDE metacommand to access them in the client forms and modules.

Variables

Variables are identifiers that are able to store data during run time. All variables must have a data type associated with them. This association can be explicit or implicit. Explicit type association occurs when the variable name is appended by one of the type-declaration characters, shown in TABLE A-1. For example, X# denotes the variable X that stores the double data type. Similarly, MyName$ is a variable that stores a string of characters. By default, a variable has the single data type. Visual Basic offers the Deftype declaration to permit you to alter the default data types. The rule of using these statements is that you specify the default type of variables by their first letter. The general syntax for using a Deftype declaration is:

```
DEFtype letter1 [-letter2] [,letter3, . . . ]
```

The Deftype declares the default types by using a single character, a range of contiguous characters, or a comma-delimited list that combines the first two kinds. Because the Visual Basic identifiers are case-insensitive, the uppercase and lowercase characters can be interchanged in a DefType without altering the outcome of the declaration. The various Deftype declarations are shown below:

```
DEFINT   Integer
DEFLNG   Long
DEFSNG   Single
DEFDBL   Double
```

```
DEFCUR  Currency
DEFSTR  String
```

Here are a few examples of the DefType declarations:

```
DEFDBL A-Z
DEFINT I-O, X
```

The first Deftype declares that all of the variables are double type by default. The Second DefType declares that the variables beginning with I to O, and with X, are integers by default. The two declarations can occur in the same program. The effect is that the second Deftype overrides the first one. The general rule is that Deftype declarations override the previous one.

Visual Basic also permits you to declare the data types of variables using the entire variable names and not just the first letter. Such declarations include common, shared, and normal (local volatile) variables. The general syntax for such declarations is:

```
DIM [SHARED] variable1 AS type1 [, variable2 AS type2, . . . ]
[COMMON] SHARED variable1 AS type1 [, variable2 AS type2, . . . ]
COMMON variable1 AS type1 [, variable2 AS type2, . . . ]
```

The SHARED declaration allows a variable in a procedure to share its value with the module level. Variables that are declared COMMON SHARED and DIM SHARED are shared between the module level and the procedures of that module. COMMON and COMMON SHARED variables can be shared among two or more modules. Each module must contain the same variable declaration. Several examples of declaring normal, shared, and common variables are shown below:

```
DIM Index AS INTEGER
DIM SHARED Z, X, Y AS DOUBLE
COMMON SHARED TheFilename AS STRING
SHARED ProgramParameter AS STRING
DIM ZipCode As STRING * 9
COMMON Wealth AS CURRENCY
```

Notice that you can only declare fixed strings using the AS STRING * size format. Using the $ type-declaration character or the DEFSTR results in declaring dynamic strings.

Arrays of variables

Arrays are data structures that are popular in the majority of programming languages. An array represents a group of variables that share the same name and data type, and are accessed by an index number. Each array you declare has a lower and an upper index range (and, therefore, a specific size). The OPTION BASE statement sets the default lower index to either 0 or 1. The general syntax for using the Option Base is:

OPTION BASE {0 | 1}

If there is no OPTION BASE statement in the program, the default OPTION BASE 0 is assumed. The general syntax for declaring an array is:

{DIM | REDIM} array([low1 TO] high1 [,[low2 TO] high2]) AS type

The above general form shows that you can explicitly declare the lower index range of an array. This is the recommended method because it makes the declaration of the array crystal clear. When you omit the lower bound and the TO keyword, the default lower range is specified by the OPTION BASE statement.

Visual Basic supports both dynamic (variable-size) and ordinary (fixed-size) arrays. The difference between the two types is that dynamic arrays can be resized at run time, while fixed arrays cannot. Therefore, choose the size of your fixed array with care. If it is hard to predict the required size of a fixed array, switch to a dynamic array.

Here are some examples of declaring arrays:

```
' arrays declared in the global module
OPTION BASE 1
DIM Names1 (10) AS STRING
DIM Names2 (1 To 10) AS STRING
DIM Names3$ (1 To 10)
DIM Names4$ (10)
```

The four arrays declared above are equivalent in the way their range indices and data types are defined. The array Names1 relies on the OPTION BASE statement to supply its default lower limit. The "AS STRING" declares the array to have STRING-typed elements. The array Names2 explicitly defines the lower and upper index ranges. Again, the "AS STRING" declares the basic array type to be STRING. The Names3 array declares the index range just like array Names2. However, the $ type-declaration character is used to indicate the basic type of the array element. Finally, the fourth array Name4 declares the array size just like Name1, and the array type just like Name3.

Additional declarations of arrays are shown below:

```
' arrays in a global module
DIM TheVars (65 TO 90) AS DOUBLE, X#(-4 TO 4)
' arrays in a module
DIM Filename$(1 TO 55), IdxArray(1 TO 55) AS INTEGER
' arrays in a subroutine
DIM SHARED SizeArray (1 TO MAX_SIZE) INTEGER ' MAX_SIZE is a
constant
```

The () operator is used in accessing the array members. Here is an example of manipulating an array:

```
DIM FactorialTable (1 TO 20) AS DOUBLE
DIM I, J AS INTEGER
FOR I = 1 TO 20
```

```
    FactorialTable(I) = 1
    FOR J = 1 TO I
        FactorialTable(I) = I * FactorialTable(I)
    NEXT J
NEXT I
```

The array FactorialTable stores the factorials of the first 20 integers. The outer FOR-NEXT loop is used to select each element of the FactorialTable array. The inner FOR-NEXT loop is used to calculate the sought factorials.

Two-dimensional arrays

Visual Basic supports two-dimensional arrays (commonly known as matrices). The declaration of matrices is merely an extension of the declaration of one-dimensional arrays. Matrices add a second index range. Examples of matrices are shown below:

```
OPTION BASE 1
DIM MainMatrix, SolutionMatrix ( 10, 1 TO 10) AS DOUBLE
DIM Mat1#( 20, 20), Mat2 (3 TO 5, 5 TO 9) AS INTEGER
DIM Appointments (1 TO 7, 9 TO 17) AS STRING
DIM Apps$ (7, 9 TO 17)
```

The () operator is also used to access the elements of a matrix. However, in this case, the () operator takes two integer-typed arguments that are separated by a comma. An example of accessing the elements of a matrix is shown below:

```
DIM Mat (1 TO MAT_SIZE, 1 TO MAT_SIZE) AS DOUBLE
DIM I, J AS INTEGER
FOR I = 1 TO MAT_SIZE
    FOR J = 1 TO MAT_SIZE
        Mat(I, J) = SQR(I * J)
    NEXT J
NEXT I
```

Multidimensional arrays

Visual Basic enables you to declare arrays that have up to 60 dimensions. From a practical point of view, the probability of using multidimensional arrays decreases significantly with the number of dimensions. One-dimensional arrays are very common; two-dimensional ones are slightly less common; three-dimensional arrays are rather rare; and four-dimensional arrays and up are very rare. The declaration and access of multidimensional arrays are very similar to the one- and two-dimensional arrays. An example for declaring and using a three-dimensional array is shown below:

```
OPTION BASE 1
    DIM Cube (1 TO CUBE_SIZE, 1 TO CUBE_SIZE, 4) AS DOUBLE
    DIM I, J, K AS INTEGER
```

```
    FOR I = 1 TO CUBE_SIZE
        FOR J = 1 TO CUBE_SIZE
            FOR K = 1 TO 4
                Cube(I, J, K) = SQR(I * J) / K
            NEXT K
        NEXT J
NEXT I
```

Dynamic arrays

Visual Basic also implements dynamic arrays. The advantage of these arrays is that you can alter both their sizes and number of dimensions during run time. As a result, dynamic arrays provide tailor-fit sizes and take the guesswork out of sizing arrays at design time. The creation of dynamic arrays is a simple process:

1. Declare the array with a DIM statement, using an empty dimension list. This step does not allocate any array elements.
2. Allocate the array elements using a REDIM statement. This statement specifies the exact dimensions of the array. The REDIM statement preserves the content of the expanded array when you include the PRESERVE keyword right after the REDIM keyword. Otherwise, the values of the elements created by a previous REDIM statement are lost. This is called destructive redimensioning.

Dynamic arrays are typically used inside procedures and functions. An example of creating a dynamic array is shown below:

```
DIM Mat () AS DOUBLE ' declare dynamic matrix
SUB ProcessMatrix ()
    .
    .
    .
    REDIM Mat (1 TO 10, 1 TO 10)
    .
    .
    .
END SUB
```

Control arrays

Visual Basic supports a new kind of array structure that is applied to controls. While control arrays share a number of properties with standard (both fixed and dynamic) arrays, they also have their differences.

Control arrays represent a group of controls that offers very similar or closely related operations. For example, consider an array of option boxes used in a text search dialog box. The control array is used to determine the search direction: forward, backward, or global.

Because control arrays contain similar individual controls, they can share the same event-handling procedures. For example, the array of the text-search option buttons can share the same SearchDirBtn_Click procedure, as shown below:

```
SUB SearchDirBtn_Click(Index AS INTEGER)
```

The Index parameter is required to select the proper control element in the array. Inside the event-handling procedure you can access the sought control using the index parameter. Thus, the controlArrayName(Index) terms accessed the Index'th control for the SearchDirBtn(Index) procedure.

Control arrays have the following features and rules:

- The control array can only be one-dimensional.
- The lowest array index is 0.
- Control arrays do not have preset upper bounds.
- A Visual Basic program can expand the control array at run time. The expansion is non-destructive, just like the REDIM PRESERVE statement used with dynamic arrays.
- Unlike standard arrays, control arrays can be sparse. This means that you can create a control array with an arbitrary sequence of indices. For example, you can have the SearchDirBtn option button array have elements that are indexed by 0, 1, 5, 9, 10, 11, and 21—the indices need not be in perfect sequence.
- Control arrays are NOT declared in the application code.
- Control arrays are created at design time by using either one of the following methods: in the first method, you assign the same name to the CtlName property of two similar controls. The second method creates a control array by assigning a value to the Index property (which is blank for controls that are members of a control array).
- You can change the setting of the Index property of a control at design time only.
- Control arrays cannot be passed as arguments to a procedure or function. However, you can pass an individual element of the control array as an argument.

User-defined data types

Visual Basic, like QuickBasic, allows you to declare your own data types. Each user-defined data type is made up of one or more elements (or fields). The general syntax for user-defined types is:

```
TYPE userType
    elementName1 AS typeName
    [elementName2 AS typeName
```

```
.
.
.
END TYPE
```

The typeName refers to the basic data types, fixed strings, static arrays, and other previously declared user-defined types.

Some examples of user-defined data types are shown below:

```
TYPE PersonRec
    LastName AS STRING * 10
    FirstName AS STRING * 6
    SSN AS STRING * 10
END TYPE

TYPE HealthRec
    Person AS PersonRec
    Weight AS SINGLE
    Age AS INTEGER
END TYPE
```

Once a user-defined data type is defined you can declare variables that have that type, using the "AS type" form. Examples of declaring variables that have user-defined types are shown next:

```
DIM Me AS PersonRec
STATIC Machismo AS HealtRec
DIM Employees(100) AS PersonRec
```

The individual fields (or elements) are accessed using the dot access operator. The general syntax for using the access operator is:

```
user_defined_type_variable.field
```

Here are some examples of accessing the data fields:

```
Me.LastName = "Shammas"
FullName$ = Me.FirstName + " " + Me.LastName
Employees(3).SSN = "333-33-3333"
    Machismo.Age = 32
Machismo.Person.LastName = "Hercules"
```

In the last example, the variable Machismo is of type HealtRec. This data type has the Person field that is of type PersonRec. This means that to access the LastName field in the Person field, the dot operator must be used twice, as shown above.

Decision-making control structures

Decision-making control structures enable you to examine conditions and take alternate actions. The Visual Basic decision-making structures are: the IF-THEN structure, the IF-THEN-ELSE structure, and the SELECT-CASE structure.

The IF-THEN structure

The IF-THEN is a single alternative decision-making control structure. There are two general syntaxes for the IF-THEN structure:

```
IF condition THEN statement

IF condition THEN
    sequence of statements
END IF
```

The IF-THEN structure examines the condition. If it is true (that is not equal to 0), the statements in the THEN clause are executed.

Examples of the IF-THEN structure are shown below:

```
IF ErrorMessageBox.Text = "" THEN ResultBox.Text =
Str$(NumericResult)

IF X# > 0 THEN
    Z# = LOG(X#)
    ResultBox.Text = STR$(Z#)
END IF
```

The IF-THEN-ELSE structure

The basic IF-THEN-ELSE is a dual-alternative control structure. The structure allows you to take alternate actions based on the tested condition. The general syntax for the basic IF-THEN-ELSE structure is:

```
IF condition THEN
    sequence of statements
ELSE
    sequence of statements
END IF
```

If the tested condition is true, the block of statements in the THEN clause is executed. Otherwise, the block of statements in the ELSE clause are executed. An example of the IF-THEN-ELSE structure is shown below:

```
IF X# > 0 THEN
        Z# = LOG(X#)
        ResultBox.Text = STR$(Z#)
ELSE
    BEEP
    ErrorMessageBox.Text = "Invalid argument for function Log"
END IF
```

The IF-THEN-ELSE structure can offer multiple-alternative decision making. This is made possible by inserting one or more ELSEIF clauses after the first Then clause. The general syntax for the extended IF-THEN-ELSE structure is:

```
IF condition1 THEN
    sequence # 1 of statements
```

```
ELSEIF condition2 THEN
    sequence #2 of statements
[ELSEIF condition3 THEN
    sequence #3 of statements]
ELSE
    default sequence of statements
END IF
```

The ELSE clause of the extended IF-THEN-ELSE structure serves as a catch-all clause. The structure tests the conditions of the IF and ELSEIF in the sequence in which they appear in the structure. When any one of these conditions is true, the statements in the corresponding Then clause are executed. Program execution resumes after the END IF. If all of the tested conditions are not true, the statements of the ELSE clause are executed. To make the extended IF-THEN-ELSE structure run faster, arrange the tested conditions in the order of likelihood of being true. This arrangement generally minimizes the number of conditions that must be evaluated.

An example of using the extended IF-THEN-ELSE structure is shown below:

```
X = VAL(Operand1Box.Text)
Y = VAL(Operand2Box.Text)
Operator$ = OperatorBox.Text
IF Operator$ = "+" THEN
    Z = X + Y
ELSEIF Operator$ = "-" THEN
    Z = X - Y
ELSEIF Operator$ = "*" THEN
    Z = X * Y
ELSEIF (Operator$ = "/") AND (Y <> 0) THEN
    Z = X / Y
ELSE
    BEEP
    ErrorMessageBox.Text = "Invalid operator or divide-by-zero error"
END IF
```

The IF-THEN-ELSE structure examines the value of a series of conditions that compare the string variable, Operator$, with literal strings that represent the basic math operations. If Operator$ contains "-", the condition of the first If is false. This causes the second condition (Operator$ = "-") to be tested. This turns out to be true, and the statement, Z = X - Y, is executed. Execution resumes after the END IF. Notice that the last tested condition uses the relational AND operator to combine two smaller tests, (Operator$ = "/") and (Y <> 0). This test prevents the division of X by Y from occurring when the value of Y is zero.

The SELECT-CASE structure

The multi-alternative IF-THEN-ELSE structure may be replaced by the SELECT-CASE structure when the tested conditions compare the same expression with different values. The general syntax for the SELECT-CASE structure is:

```
SELECT CASE expression
    CASE expressionList1
        sequence #1 of statements
    [CASE expressionList1
        sequence #2 of statements]
    [CASE ELSE
        sequence #n of statements]
END SELECT
```

The expression examined by the SELECT-CASE statement can be a variable, a function, or a Basic expression involving constants, variables, and functions. The CASE clauses contain a comma-delimited list of values that are compared against the examined expression. This means that the expression and the data in the expression list must have matching, or at least compatible, data types. The types of basic items that appear in an expression list are:

- A single value.
- A range of values (used for numeric types) that use the following general syntax:

  small TO big

A relational operator with a value. This compares the examined expression with that value, using the relational operator. The general syntax is:

IS relational_operator value

The values mentioned in the above basic items can be literal constants, symbolic constants, or variables.

The following simple example of the SELECT-CASE structure is one that modifies the last example of the IF-THEN-ELSE structure:

```
X = VAL(Operand1Box.Text)
Y = VAL(Operand2Box.Text)
Operator$ = OperatorBox.Text
SELECT CASE Operator$
    CASE "+"
        Z = X + Y
    CASE "-"
        Z = X - Y
    CASE "*"
        Z = X * Y
```

```
    CASE "/"
        IF Y <> 0 THEN Z = X / Y
    ELSE
        BEEP
        ErrorMessageBox.Text = "Invalid operator"
END SELECT
```

The examined expression in the above SELECT-CASE structure is the value of the string variable Operator$. Each CASE clause provided a single-item expression list of values to compare with Operator$. If Operator$ contains "–" it will match the expression list of the second CASE clause, and the statement, Z = X – Y, is executed. Execution resumes after the END SELECT. The above example can be expanded to include multivalue expression lists. The next example also compares the Operator$ with strings that contain the names of the sought operations (in long and short form). The examined expression is actually a function, namely, UCASE$(Operator$). This allows the strings that describe the operations to be in uppercase, with the benefit that the Select-Case structure can respond to the name of the sought operator typed in any letter case combination, such as Multiply, multiply, and even mUlTiPlY!

```
X = VAL(Operand1Box.Text)
Y = VAL(Operand2Box.Text)
Operator$ = OperatorBox.Text
SELECT CASE UCASE$(Operator$)
    CASE "+", "ADD"
        Z = X + Y
    CASE "–", "SUBTRACT", "SUB"
        Z = X – Y
    CASE "*", "DIVIDE" ,"DIV"
        Z = X * Y
    CASE "/", "MULTIPLY", "MUL"
        IF Y <>0 THEN Z = X / Y
    ELSE
        BEEP
        ErrorMessageBox.Text = "Invalid operator"
END SELECT
```

The third example of the SELECT-CASE structure involves a numeric expression. It shows the use of the three basic items of a CASE expression list:

```
SELECT CASE DangerLevel%
    CASE 0, 4, 6, 10 TO WARNING1, IS > CriticalLimit%
        MessageBox.Text = "Case 1"
    CASE 1 TO 3, 5, 7, WARNING1+1 TO WARNING2, CriticalLimit%
        MessageBox.Text = "Case 2"
    CASE 8 TO 9, WARNING2+1 TO CriticalLimit%-1
        MessageBox.Text = "Case 3"
    CASE ELSE
```

```
        FOR I% = 1 TO 10
            BEEP
        NEXT I%
        MessageBox.Text = "Reactor core melt-down!"
    END SELECT
```

The WARNING1 and WARNING2 identifiers are symbolic constants (that I wrote in uppercase letters). The CriticalLimit% is an integer variable.

Loop control structures

Loops are structures that are used for iterating one or more statements. There are three types of loops, all supported by Visual Basic. These are the fixed-iteration loops, conditional loops, and open loops. As their names suggest, the fixed-iteration loops iterate for a fixed number of times; the conditional loops iterate while or until a tested condition is true; and the open loop iterates indefinitely. Visual Basic offers the FOR-NEXT fixed-iteration loop, the DO-WHILE-LOOP and DO-UNTIL-LOOP conditional loops, and the DO-LOOP open loop.

The FOR-NEXT loop

The FOR-NEXT loop is the oldest and most popular loop in any Basic implementation. The general syntax for the FOR-NEXT loop is:

```
FOR loopCounter = start TO finish [STEP increment]
    sequence of statements
    [EXIT FOR]
    [sequence of statements]
NEXT [loopCounter [,loopCounter]]
```

The loop counter loopCounter is a variable that is used to keep track of the number of iterations. There are four possible general loop settings:

```
FOR loopCounter = smallStart TO bigFinish
FOR loopCounter = smallStart TO bigFinish STEP increment
FOR loopCounter = bigStart TO smallFinish STEP -1
FOR loopCounter = bigStart TO smallFinish STEP decrement
```

The first two forms of the FOR-NEXT loop are upward-counting loops, where smallStart is less than or equal to bigFinish. In the first form, the loop control is incremented by 1, the default value. The second form shows the STEP clause with an increment other than 1. (It can be greater or less than 1. You can even use STEP 1.) The number of iterations for the first form of the FOR-NEXT loop is equal to (bigFinish − smallStart + 1). If this number is negative, the FOR-NEXT loop does not iterate (that is, there is no such thing as a negative number of iterations). The number of iterations for the second form is equal to ((bigFinish − smallStart + 1) / increment).

The third and fourth forms of the FOR-NEXT loops are downward-

counting loops, where bigStart is greater or equal to smallFinish. In the third form, the loop control is decremented by 1 (conversely, it can be said that the loop counter is incremented by −1). In the fourth form, the STEP has an increment with a negative value other than −1 (you can say that the third form is a special case of the fourth form). The number of iterations for the third form of the FOR-NEXT loop is equal to (bigStart − smallFinish + 1). The number of iterations for the fourth form is equal to ((bigStart − smallFinish + 1) / −decrement).

If the value of the STEP increment or decrement is 0, the FOR-NEXT loop iterates indefinitely.

The EXIT FOR statement allows you to exit the current FOR-NEXT loop and resume execution after its Next statement. More about exiting loops in the "Exiting from Loops" section.

The NEXT statement can be followed by the name of the loop control. In the case of nested loops that logically end at the same location, you can use a single NEXT statement followed by a list of the loop controls for the nested loops.

Examples for various forms of the FOR-NEXT are shown below:

```
FOR I% = 1 TO 100
    A#(I%) = 2 * I%
NEXT

FOR I% = 34 TO 55 STEP 2
    Sum = Sum + Value#(I%)
NEXT I%

FOR I% = 23 TO 4 STEP -1
    A#(I%) = SQR(I%)
NEXT I%

FOR I% = 23 TO 4 STEP -2
    A#(I%) = SQR(I%)
NEXT I%
```

The following nested loops are equivalent, even though the second loop uses a single NEXT statement:

```
FOR I% = 1 TO 10
    FOR J% = 1 TO 10
        DataMatrix#(I%, J%) = 0
    NEXT J%
NEXT I%

FOR I% = 1 TO 10
    FOR J% = 1 TO 10
        DataMatrix#(I%, J%) = 0
NEXT J%, I%
```

The DO-WHILE-Loop

The DO-WHILE-Loop is a conditional loop that iterates based on a tested condition. Visual Basic implements two forms of the DO-WHILE-Loop, which I name the "leading" and the "trailing." The general syntaxes for the two DO-WHILE loops are:

```
' the "leading" DO-WHILE loop
DO WHILE condition
    sequence of statements
    [EXIT DO]
    sequence of statements
LOOP
```

and,

```
' the "trailing" DO-WHILE loop
DO
    sequence of statements
    [EXIT DO]
    sequence of statements
LOOP WHILE condition
```

The leading DO-WHILE loop is much more common than the trailing version. The leading DO-WHILE loop is also equivalent to the WHILE-WEND loop that is available in BASICA, GW-BASIC, and QuickBasic. The leading DO-WHILE loop iterates while a condition is true. This means that if the condition is false the first time it is tested, the leading DO-WHILE loop does not iterate. Instead the program execution resumes after the Loop statement.

The trailing DO-WHILE loop examines the condition at the end of the loop. This guarantees that the loop iterates at least once.

The EXIT DO statement enables the program to exit the current loop and resume after the LOOP statement.

Examples of using either form of the DO-WHILE loop are shown below:

```
' remove the leading spaces from a string
L% = LEN(S$)
DO WHILE (L% > 0) AND (MID$(S$, 1, 1) = " ")
    S$ = MID$(S$, 2, L%)
    L% = L% - 1
LOOP
```

The above code uses a leading DO-WHILE loop to trim the string S$ from leading spaces. The condition tested is made up of two subconditions. The first subcondition tests if L% is positive (that is, the length of the string is not zero). This serves two purposes: it detects if string S$ is either originally empty (i.e., null) or has become empty in the course of the iteration (the original S$ contained nothing but spaces).

The second example is a trailing DO-WHILE loop that obtains the square root of a variable, X#, by applying Newton's method. The loop iterates

while the difference between X# and the square of the variable Guess# (the guess for the square root) exceeds some acceptable small number:

```
' find the square root of X# using Newton's algorithm
Guess# = X# / 2
DO
    IF Guess# <= 0 THEN EXIT DO
    Guess# = 0.5 * (X# / Guess# + Guess#)
LOOP WHILE ABS(X# - Guess# ^2) > .000001
```

The DO-UNTIL-Loop

The DO-UNTIL-Loop is another conditional loop that iterates based on a tested condition. Again, Visual Basic implements two forms of the DO-UNTIL-Loop, which I name the "leading" and the "trailing". The general syntaxes for the two DO-UNTIL loops are:

```
' the "leading" DO-UNTIL loop
DO UNTIL condition
    sequence of statements
    [EXIT DO]
    sequence of statements
LOOP
```

and,

```
' the "trailing" DO-UNTIL loop
DO
    sequence of statements
    [EXIT DO]
    sequence of statements
LOOP UNTIL condition
```

The trailing DO-UNTIL loop is much more common than the leading version. The leading DO-UNTIL loop iterates until a condition is true. This means that if the condition is true the first time it is tested, the leading DO-UNTIL loop does not iterate. Instead the program execution resumes after the Loop statement.

The trailing DO-UNTIL loop examines the condition at the end of the loop. This guarantees that the loop iterates at least once.

The EXIT DO statement enables the program to exit the current loop and resume after the LOOP statement.

Examples of using either form of the DO-UNTIL loop are shown below:

```
' remove the leading spaces from a string
L% = LEN(S$)
DO UNTIL (L% = 0) OR (MID$(S$, 1, 1) <> " ")
    S$ = MID$(S$, 2, L%)
    L% = L% - 1
LOOP
```

The above code uses a leading DO-UNTIL loop to trim the string S$ from leading spaces. The condition tested is made up of two subconditions. The first subcondition tests if L% is zero (that is, the length of the string is zero). This accomplishes two tasks: it detects if string S$ is either originally empty (i.e., null) or has become empty in the course of the iteration (the original S$ contained nothing but spaces).

The second example is a trailing DO-UNTIL loop that obtains the square root of a variable, X#, by applying Newton's method. The loop iterates until the difference between X# and the square of the variable Guess# (the guess for the square root) is less than or equal to some acceptable small number:

```
' find the square root of X# using Newton's algorithm
Guess# = X# / 2
DO
    IF Guess# <= 0 THEN EXIT DO
    Guess# = 0.5 * (X# / Guess# + Guess#)
LOOP UNTIL ABS(X# - Guess# ^2) <= 0.000001
```

The DO-LOOP

Visual Basic offers the open Do-Loop. In principle, this loop iterates indefinitely. While this aspect sounds like the program is asking for trouble, the DO-LOOP invariably tests for a condition inside the loop body and uses the EXIT DO statement to terminate the iteration. The general syntax for the DO-LOOP is shown below:

```
DO
    sequence of statements
    [IF exit_condition THEN EXIT DO]
    sequence of statements
LOOP
```

As you can see, add a While or Until clause at either end of the open loop and you end up with DO-WHILE or DO-UNTIL loops.

An example of using a Do-Loop is shown below:

```
Sum# = 0
I% = 1
DO
    Term# = I%^2 / Factorial(I%)
    IF Term# < 0.00001 THEN EXIT DO
    Sum# = Sum# + Term#
    I% = I% + 1
LOOP
```

The above loop calculates a summation series. Each term of the series is a function of the variable I%. As more terms are added, the value of each term decreases. The IF statement, inside the loop, tests if the calculated term is too small, and accordingly exits the loop. If the value of the term is

still significant, the remaining loop statements are executed and the loop reiterates.

Exiting from loops

Describing the normal use of the open DO-LOOP earlier included a discussion about using the EXIT DO statement. While the EXIT DO plays a key role in using the DO-LOOP, it can still be used with the DO-WHILE and DO-UNTIL conditional loops. Similarly, the EXIT FOR statement can be used with the FOR-NEXT loops to perform an "early" exit or loop termination. Using EXIT DO and EXIT FOR provides Visual Basic programming with powerful constructs that rival more structured languages like Pascal and C.

The following code example is a modified version of the DO-LOOP example. A FOR-NEXT loop is used to calculate the summation series. Notice that the loop control, I%, ranges from 1 to 32767 (the upper limit of integers). In reality, the loop will iterate a few times before the tested condition invokes the EXIT FOR:

```
Sum# = 0
FOR I% = 1 TO 32767
    Term# = I%^2 / Factorial(I%)
    IF Term# < 0.00001 THEN EXIT FOR
    Sum# = Sum# + Term#
NEXT I%
```

Nesting control structures

Visual Basic empowers you to write code that nests the various control structures. The Exit statements can serve to cleverly exit some or all of these nested control structures. An example of nesting control structures is given below—the code of a rather elaborate matrix initialization:

```
FOR I% = 1 TO MatSize%
    FOR J% = 1 TO MatSize%
        SELECT CASE I%
            CASE 1 TO 10
                Mat#(I%, J%) = I% + J%
            CASE 11 TO 20
                Mat#(I%, J%) = I% * J%
            CASE 21, 35, 65
                IF J% > 5 THEN
                    Mat#(I%, J%) = I% + 2 * J%
                ELSE
                    Mat#(I%, J%) = 0
                END IF
            CASE ELSE
                Mat#(I%, J%) = SQR(I% * J%)
        END SELECT
```

```
    NEXT J%
NEXT I%
```

SUB and FUNCTION procedures

Visual Basic implements three types of routines, namely, event-handling procedures, SUB procedures, and FUNCTION procedures. Event-handling procedures are actually special SUB procedures that handle specific events for certain objects. The SUB and FUNCTION procedures (including event-handling procedures) can call each other, as long as they are all located in the same form or module.

The event-handling procedures are SUB procedures that associate an object with an event. The names of these procedures are derived from the names of related object and events. The general syntax for an event-handling procedure is:

```
SUB objectName_eventName([parameter list])
    sequence of statements
End SUB
```

When the program is running and you generate an event, the object where the event has occurred, such as a mouse click, recognizes the event and attempts to respond to that event. This attempt is translated into the following steps:

1. Locate the procedure that has the name of the responding object and the occurring event.
2. If the sought procedure exists, it is automatically called. When the event-handling procedure finishes working, the event is considered handled.
3. If the sought procedure does not exist, the event is abandoned.

Event-handling procedures are very similar to ordinary SUB procedures discussed next.

SUB procedures

The general syntax for a Sub procedure is:

```
SUB procedureName([parameter list])
    sequence of statements
END SUB
```

The parameter list contains a comma-delimited list of parameters that determine the amount and type of information that is passed to the SUB procedure. When a SUB procedure is invoked, a set of arguments is supplied to match the parameter list. The general syntax for each parameter is:

[BY VAL] parameterName [()] [AS type]

The BY VAL option causes an argument (the value passed to a parameter when the procedure is called) to be passed by value. This means that a copy of the argument is submitted to the procedure. Consequently, any changes made to that argument inside the procedure do not affect the original argument. By default, arguments are passed by reference, unless they are expressions. Passing an argument by reference means that the procedure accesses the address of the argument. Consequently, any changes made to the argument inside the procedure affect that argument outside the procedure. Enclosing a variable in parentheses makes the argument an expression and passes its data by value. Passing arguments by reference offers a two-way communication of data between the procedure and its caller.

The parameter list may include arrays. To indicate that the parameter is an array, include the parentheses after the parameter name.

The data type of the parameters can be specified in one of two ways. The first method uses the type-declaration characters after the parameter name (!, @, #, $, %, or &). This method works only for the predefined data types. The second method uses the As type clause. This method must be used in declaring a parameter that is of a user-defined type. When declaring a parameter to be of a standard type, use one of the following names: INTEGER, SINGLE, DOUBLE, LONG, CURRENCY, or STRING. In addition, Visual Basic allows you to pass special types, namely, FORM and CONTROL. These parameters enable you to use the procedure to set and query the properties (when applicable) of the forms and control objects at run time.

Examples of Sub procedures are shown below:

```
OPTION BASE 1
CONST True = -1, False = 0
SUB ShellSort(A() AS STRING, BY VAL N AS INTEGER, InOrder AS
    INTEGER)
' Shell sort method
DIM Skip, I, J AS INTEGER
DIM Temp AS STRING
Skip = N
DO WHILE Skip > 1
    Skip = Skip \ 2
    InOrder = True
    DO
        FOR J = LBOUND(A,1) TO UBOUND(A,1) + N - 1 - Skip
            I = J + Skip
            IF A(J) < A(I) THEN
                Temp = A(I)
                A(I) = A(J)
                A(J) = Temp
                InOrder = False
            END IF
```

```
            NEXT J
        LOOP UNTIL InOrder
    LOOP
    END SUB

    SUB BinSearch(BY VAL Find$, A() AS STRING, BY VAL N%, Index%)
    ' binary search in ordered arrays
    DIM Hi, Lo, Median AS INTEGER
    Lo = LBOUND(A, 1)
    Hi = LBOUND(A, 1) + N - 1
    DO
        Median = (Lo + Hi) \ 2
        IF Find$ < A(Media) THEN
            Hi = Median - 1
        ELSE
            Lo = Median + 1
        END IF
    LOOP UNTIL (Find$ = A(Median) OO (Lo > Hi)
    IF Find$ = A(Median) THEN
        Index% = Median
    ELSE
        Index% = LBOUND(A, 1) - 1
    END IF
    END SUB

    SUB SearchData(BY VAL Find$, A() AS STRING, BY VAL N%,
        InOrder%, Index%)
    ' search for Find$ in array A()
    IF NOT InOrder% THEN
        ShellSort A, N%, InOrder%
    END IF
    BinSearch Find$, A, N%, Index%
    END SUB
```

The ShellSort procedure sorts the first N elements of the array, using the Shell-Metzner algorithm. The lower bounds of the array are not restricted to 0 or 1. The BinSearch procedure performs a binary search on the ordered array A. The first N elements are involved in the search. The SearchData procedure is a more general search procedure. It examines the value of the argument for InOrder. If that argument is not false (that is true), it calls the ShellSort procedure to sort the array. The arguments A and InOrder% are passed by reference, allowing the ShellSort to change their values. By contrast, the argument N% is passed by value (as specified in the declaration of the parameter list). Once the array is sorted, the BinSearch procedure is called. The arguments Find$ and N% are passed by value (as specified in the declaration of the parameter list), while the arguments A and Index% are passed by reference. The Index% argument returns the index of the matching element or (LBOUND(A,1)-1) if no match is found. It is highly recommended that arrays be passed by reference, to save memory.

Passing by value means that the array is copied in memory. Since this process can be repeated in chained procedure calls, you may easily run out of memory when handling large arrays.

FUNCTION procedures

The general syntax for FUNCTION procedures is:

```
FUNCTION functionName([parameter list]) [AS type]
    sequence of statements
    functionName = functionResult
    [sequence of statements]
END FUNCTION
```

The FUNCTION procedures resemble the Sub procedures in many aspects. The main difference is that the FUNCTION procedure, as the name might suggest, returns a value. Secondary differences between the Sub and the FUNCTION procedures include:

- When the function has no parameters, you must include empty parentheses with each function call.
- The data type of the function result is declared in one of two ways. The first uses the type-declaration character that is appended to the function's name. The second method employs the "AS type" clause. The first method works only for the basic data types. The second method must be used with user-defined data types.
- The function result is obtained by assigning an expression to an identifier that matches the function's name.

Some examples of FUNCTION procedures are shown next. The functions BinSearch and SearchData are modified versions of the Sub procedures with the same name, shown earlier. Notice that for both functions, the search result is not returned via an argument passed by value. Instead, the sought result is returned using the function's names:

```
FUNCTION BinSearch(BY VAL Find$, A() AS STRING, BY VAL N%) AS
    INTEGER
' binary search in ordered arrays
DIM Hi, Lo, Median AS INTEGER
Lo = LBOUND(A, 1)
Hi = LBOUND(A, 1) + N - 1
DO
    Median = (Lo + Hi) \ 2
    IF Find$ < A(Media) THEN
        Hi = Median - 1
    ELSE
        Lo = Median + 1
    END IF
LOOP UNTIL (Find$ = A(Median) OR (Lo > Hi)
IF Find$ = A(Median) THEN
```

```
    BinSearch = Median
ELSE
    BinSearch = LBound(A, 1) - 1
END IF
END SUB

FUNCTION SearchData%(BY VAL Find$, A() AS STRING, BY VAL N%,
    InOrder%)
    ' search for Find$ in array A()
IF NOT InOrder% THEN
    ShellSort A, N%, InOrder%
END IF
SearchData% = BinSearch(Find$, A, N%)
END FUNCTION
```

The data type of the result for the BinSearch function is declared using the AS clause. By contrast, the data type of the result emitted by function SearchData is declared using the type-declaration character, %. Both functions return an integer result.

Exiting procedures

The SUB and FUNCTION procedures can be exited using the EXIT SUB and EXIT FUNCTION statements, respectively. These EXIT statements return control to the caller. Examples of using the EXIT SUB and EXIT FUNCTIONS are shown below. The first procedure is a function that calculates factorials. If the value assigned to the parameter N is less than 2, assign a 1 to the function result and exit the function. Otherwise, calculate the factorial using a FOR-NEXT loop, as shown below:

```
FUNCTION Factorial (N AS INTEGER) AS DOUBLE
    DIM Product AS DOUBLE
    DIM I AS INTEGER
    ' when N is less than 1, exit function
    IF N < 2 THEN
        Factorial = 1 ' assign function result
        EXIT FUNCTION ' exit from here
END IF
' calculate factorial using a FOR-NEXT loop
Product = 1
FOR I = 1 TO N
        Product = Product * I
        NEXT I
        Factorial = Product ' assign function result
END FUNCTION
```

The next SUB procedure calculates the statistical mean and standard deviation, given the number of observations (Sum#), the sum of observations (SumX#), and the sum of the squares of the observations (SumXX#). The mean and standard deviation are returned via the parameters Mean# and Sdev#. The procedure verifies whether the value of parameter Sum# is

less than 2. If the test is positive, the procedure exits; otherwise, the sought statistics are calculated.

```
SUB CalcMeanSdev(BY VAL Sum#, BY VAL SumX#, BY VAL
SumXX#,      Mean#, Sdev#)
    ' calculate the mean and standard deviation
        Mean# = 0
    Sdev# = 0
    IF Sum < 2 THEN EXIT SUB
    Mean# = SumX# / Sum#
    Sdev# = SQR((SumXX# - SumX# ^2 / Sum#) / (Sum# - 1))
END SUB
```

Static local variables

You can declare local variables using the Static statement instead of the Dim statement. The Static declaration results in storing the local variable in a fixed memory location, allowing the variable to retain its data between calls. Normally, local variables are erased once their host procedure returns control to the caller.

An example of using static variables is shown below. The ToggleButton_Click procedure toggles the case of the string in the TextBox text box. The static local variable Is_UCase is declared to keep track of the upper/lowercase state. As a static variable, Is_UCase is able to retain its data between procedure calls and properly toggle the case of the text box. If the IS_UCase variable is instead declared with a Dim statement, the event-handler will always change the string in the text box to uppercase:

```
SUB ToggleButton_Click ()
    STATIC Is_UCase AS INTEGER
    IF TextBox.Text = "" THEN
        TextBox.Text = "Hello World!"
    ELSE
        IF Is_UCase = 0 THEN
            Is_UCase = 1
            TextBox.Text = UCASE$(TextBox.Text)
    ELSE
        Is_UCase = 0
        TextBox.Text = LCASE$(TextBox.Text)
        END IF
    END IF
END SUB
```

You can declare the local variables of a procedure to be static. This is performed by placing the STATIC keyword before the SUB keyword. The local variables can then be explicitly declared with STATIC or DIM statements. The next example is a version of the ToggleButton_Click procedure that declares its local variables Is_UCase and Counter as static.

```
STATIC SUB ToggleButton_Click ()
    DIM Is_UCase AS INTEGER
DIM Counter AS INTEGER
    Counter = Counter + 1
    ' when Counter is 20 display a special string in the text box
    IF Counter = 20 THEN
        TextBox.Text = "Surprise!"
        Count = 1
        EXIT SUB
END IF
IF TextBox.Text = "" THEN
    TextBox.Text = "Hello World!"
ELSE
    IF Is_UCase = 0 THEN
        Is_UCase = 1
        TextBox.Text = UCASE$(TextBox.Text)
    ELSE
        Is_UCase = 0
        TextBox.Text = LCASE$(TextBox.Text)
    END IF
    END IF
END SUB
```

* Form arguments

One of the new aspects of the Visual Basic language is the ability to declare Forms and controls as parameters to procedures. The FORM and the CONTROL keywords represent special data types that can be applied to procedural parameters. For example, the next SUB procedure enables you to set a new caption and display that caption in underlined bold characters:

```
SUB SetFormCaption(AnyForm AS FORM, NewCaption AS STRING)
    AnyForm.Caption = NewCaption
    AnyForm.FontBold = -1
    AnyForm.FontUnderline = -1
END SUB
```

As the code for the above procedure shows, you can assign new values to the properties of the form. Visual Basic does not allow you to assign data to the form itself (as in copying forms). Moreover, Visual Basic does not permit local variables to be declared with the type FORM.

An application that contains the SetFormCaption procedure can issue the following calls:

```
SetFormCaption "Testing . . . 1, 2, 3"
SetFormCaption "Bonjour!"
```

* Control arguments

Visual Basic enables you to declare parameters that have the CONTROL type. This feature is even more powerful than passing FORM-typed parameters, because the CONTROL parameter can represent any kind of control. An example of a simple procedure that manipulates control is shown next:

```
SUB ToggleEnableState (AControl AS CONTROL)
    AControl.Enabled = NOT AControl.Enabled
END SUB
```

The ToggleEnableState procedure toggles the Enabled property of the CONTROL-typed argument. In fact, this procedure works with all of the Visual Basic objects, except Clipboard, Debug, Printer, and Screen. You can invoke the ToggleEnabledState procedure with various control objects:

```
ToggleEnabledState Text1
ToggleEnabledState Command1
ToggleEnabledState Text2
ToggleEnabledState List1
```

Because the properties of a control depend on the kind of control, you need to exercise care in coding procedures that have Control parameters. A run time error is generated if such a procedure attempts to change a nonexistent property of a Control-typed argument. Here is an example. Consider the following procedure that alters the caption of a control:

```
SUB SetCaption(AControl AS CONTROL, NewCaption AS STRING)
    AControl.Caption = NewCaption
```

Issuing SetCaption calls like the ones shown below cause no error, since the controls passed as arguments do have a Caption property:

```
SetCaption Check1 "Check it Out!"
SetCaption Check2 "Alright!"
SetCaption Box1 "FM/AM"
```

By contrast, attempting to alter the caption of controls like text boxes, directory lists boxes, and ordinary list boxes results in a run time error because these controls lack the Caption property:

```
SetCaption Text1 '———————— Error!
SetCaption List1 '———————— Error!
```

There are two solutions for the above problem. The first one uses error-trapping techniques to handle the error. For example, the SetCaption procedure can be recoded to trap the error and simply exit:

```
SUB SetCaption(AControl AS CONTROL, NewCaption AS STRING)
    ON LOCAL ERROR GOTO NoCaption ' set error trap
    AControl.Caption = NewCaption
    ON ERROR GOTO 0 ' disable error
    EXIT SUB ' exit OK
```

```
    NoCaption:
        RESUME ExitSub
    ExitSub:
END SUB
```

The second solution tackles the problem in a more radical manner. This technique uses the special TYPEOF-IS Boolean operator to discern the exact type of control. The general syntax for using the TYPEOF-IS operator is:

```
{ IF | ELSEIF } TYPEOF control IS controlType THEN
```

Visual Basic does not allow any other Boolean or relational operator to appear with the TYPEOF-IS operator. Visual Basic uses the following keywords for the controlType:

CheckBox	Frame	PictureBox
ComboBox	HScrollBar	TextBox
CommandButton	Label	Timer
DirListBox	ListBox	VScrollBar
DriveListBox	Menu	
FileListBox	OptionButton	

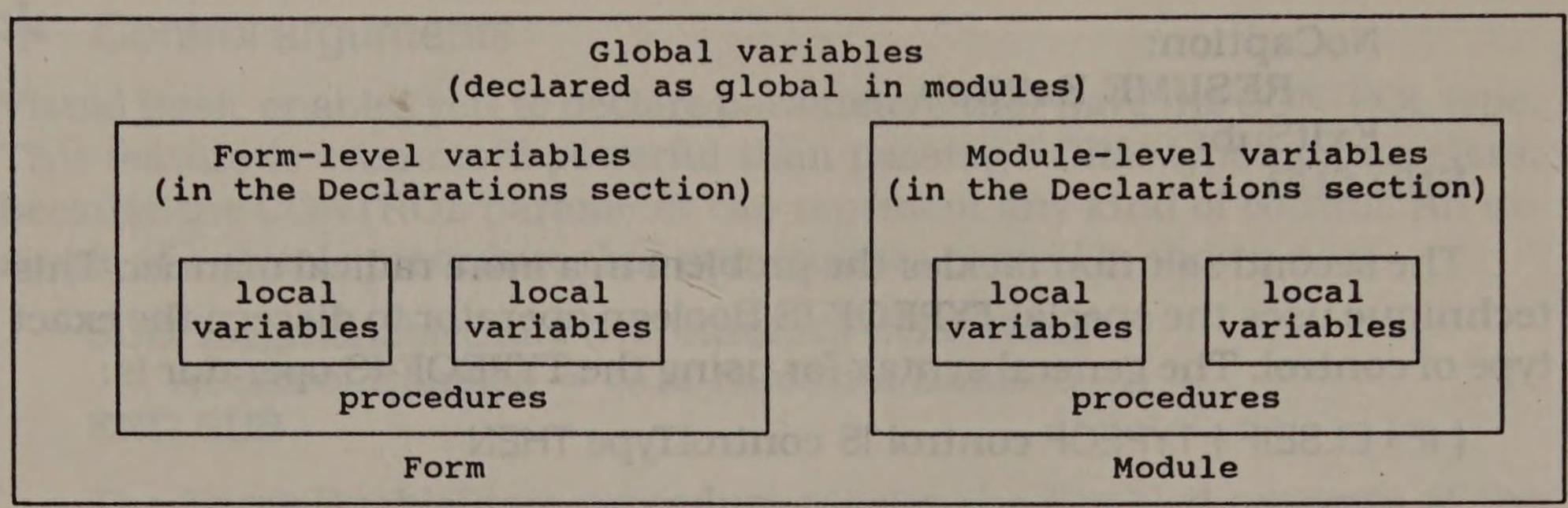
Using the TYPEOF-IS operator I can rewrite the SetCaption procedure to work with check and option boxes only. Passing controls of other types results in no action taken and no run time error:

```
SUB SetCaption(AControl AS CONTROL, NewCaption AS STRING)
    IF TYPEOF AControl IS CheckBox THEN
        AControl.Caption = NewCaption
    ELSEIF TypeOf AControl IS OptionBox THEN
        AControl.Caption = NewCaption
    END IF
END SUB
```

Modules

Modules are program libraries that export supporting SUB and FUNCTION procedures to other modules or to the various Visual Basic forms. A module passes information to the other modules or forms using either global variables or arguments of procedures. Global variables are declared as COMMON SHARED in the involved modules and forms, making them accessible to all these modules and forms. Consequently, then can pass data between modules and forms. Argument lists offer another, more controlled mechanism for passing data.

* **Scope of variables** As an implementation of Basic, Visual Basic has surely come a long way from the earlier implementations of Basic that were available up to the mid 1980s. These implementations lacked much structure in both code and data. For one, all variables were global! In addition, SUB and FUNCTION procedures were not available then, making the use of GOSUBs more prone to side effects (due to the fact that all of the variables



A-1 The different levels of variables

were global). This, among other reasons, made many programmers look down upon Basic as an inferior and toy programming language.

The advent of QuickBasic began to address the deficiencies of the earlier Basic implementations. QuickBasic introduced more structured code and data types. As the result of implementing SUB and FUNCTION procedures, multilevel data scoping was introduced, making certain variables accessible only at specific program parts.

Visual Basic has extended the levels of scoping variables. Figure A-1 shows the various levels of accessing variables. These levels are:

Global variables All of the variables declared in the global module of a Visual Basic application are recognized and accessed by all the program components. Global variables must be declared with the COMMON SHARED statement, as shown in the following examples:

```
COMMON SHARED StatusCode AS INTEGER, StatusMessage AS
STRING
```

It is recommended that global variables be kept to a minimum. Usually, global variables are used to exchange information that is required by various procedures in different forms and modules.

Form and module variables The variables declared in the general declarations section of a form are accessible by all of the procedures in that form. This enables these forms to share common information. Form-level variables are declared using the DIM SHARED statement, as shown in the example below:

```
DIM SHARED TheFormState AS INTEGER, InternalMessage AS
STRING
```

The variables appearing in the declarations section of a module are also visible to all of the procedures in that module. Form-level and module-level variables are very similar. They share similar purposes and data exchange mechanisms, and are declared in the same way.

Procedure (local) variables All variables declared inside an event-handling, normal SUB, or FUNCTION procedure are recognized only inside that procedure. You can declare local variables explicitly by using a DIM statement, or implicitly by simply using them. The latter is discouraged because you might be using the name of another variable that is declared in a higher level—you will most likely change the data in the higher-level variable. Using the DIM statement ensures that the variable you declare overrides any higher-level variable with the same name.

An example of local variables is shown in the event-handler ReverseButton_Click, listed below:

```
SUB ReverseButton_Click ()
    DIM S AS STRING
    DIM I, L AS INTEGER
    IF TextBox.Text = "" THEN
        TextBox.Text = "Hello World!"
    ELSE
        L = LEN(TextBox.Text)
        S = ""
        FOR I = 1 TO L
            S = S + MID$(TextBox.Text, L + 1 - I, 1)
        NEXT I
        TextBox.Text = S
    END IF
END SUB
```

The variables S, I, and L are declared local to the procedure. These variables are recognized only in the ReverseButton_Click procedure and override any variables with the same names that are declared in the global or form levels.

The general rule for resolving the access of different variables with the same name, declared at different levels, is this: the variable with the lowest-level is the most accessible. Thus, for example, local variables override variables with the same names, declared in the host form or host module.

B

The companion disk

This book has a companion disk that contains the .MAK, .FRM, .BI, .BAS, and other files that are mentioned in this book. These files are located in the self-extracting archive file VBCODE.EXE. To install and unpack these files perform the following steps:

1. Select the \VBDOS directory where your copy of Visual Basic is located. If you use a different name or different directory to store Visual Basic, then select that directory.
2. Create the subdirectory EASYVB by executing the following DOS command:

   C> MD EASYVB

3. Select the EASYVB subdirectory by executing the following DOS command:

   C> CD EASYVB

4. Insert the companion disk in drive A:.
5. Unpack the self-extracting archive file by executing the following command from DOS:

   C< A:VBCODE

The above command should unpack the archive file and install the files in directory \VBDOS\EASYVB.

Index

A

access keys, 78-79
  assigning to menus, 98
Add File menu option, 105
AddBtn_Click, 264
AddFileCom_Click, 105, 235
AppendCom_Click, 204
applications
  beginning programming, 15-27
  steps before programming, 29-30
Archive property, 225
arguments
  control, 332-333
  form, 331
arrays
  control, 102-105, 312-313
  multidimensional, 311-312
  two-dimensional, 311
  variable, 309-312
ASCII
  using KeyAscii, 126-127
  viewing text files, 70-76

B

BinSearch function, 232, 301
Bool function, 229
Box_Change, 264
ButtonBtn_Click, 194

C

Calendar form, 168-176
Cancel command button, 92
Cancel property, 69
CancelBtn_Click, 235
Caption text box, 92
Check boxes, 3, 32, 92, 137-140
Checked check box, 92
CleanUp, 265-266
CLEAR method, 199-200
ClearCom_Click, 183, 204
ClearScreen, 82
ClearTextBoxes, 264
CLIPBOARD, using, 197-204
CLOSE statement, 242
CLS method, 164
coding
  applications, 53-54
  attaching to controls, 57-64
  inserting in CONSTANT.BI file, 57
  window, 56-57
Combo boxes, 4, 32, 140-144
Command buttons, 3, 25-27, 31, 69
  setting access keys to, 78-79
  running the application, 64
  updating, 153-162
CompileBtn_Click, 286-288
constants
  literal, 305-306
  symbolic, 308
contacts database application, 259-273
control arrays, 102-105
  in menus, 104-105
  menus for file viewer, 105
ControlCom_Click, 184
control panel, 4-6
controls, 3-4
  aligning, 38
  altering properties of, 18-19
  arguments, 332-333
  arrays, 312-313
  attaching code to, 53-65
  changing names, 18-19
  changing values, 43-51
  deleting multiple, 38
  deleting single, 37
  dragging/dropping, 190-196
  drawing, 34-35
  drawing in the form, 17-18
  file system, 223-240
  manipulating tab order, 79-80
  moving multiple, 37
  moving single, 37
  resizing, 36-37
  timer, 207-211
CopyCom_Click, 204
countdown timer, 22-25
Ctl text box, 92
CurrentX property, 164
CurrentY property, 164
CutCom_Click, 204

D

data types, 305
  user-defined, 313-314
database, contacts, 259-273

date and time, viewing, 16-21
DaysInMonth, 174
Debug menu, 5
Default property, 69
Delete command button, 92
Delete File menu option, 105
DeleteFileCom_Click, 109-110
deleting
  multiple controls, 38
  single controls, 37
DelRecChk_Click, 264-265
dialog boxes, 111, 150-153
DIM statement, 58
DIR$ function, 243
directories
  creating, 15
  selecting, 224-225
  tree viewer, 226-232
Directory list box, 32, 224-225
disk drives, selecting, 223-224
DiskDrv_Change, 229, 235
DO-LOOP, 6-8, 323-324
DO-UNTIL loop, 322-323
DO-WHILE loop, 321-322
documentation
  custom values, 48-51
  forms, 38-40
Done command button, 92
DOS, and OOP, 1-2
DosBtn_Click, 75
DragButtonBtn_Click, 194
DragDrop event, 191-192
DragOver event, 190-191
DragTextBtn_Click, 195
drawing
  controls, 34-35
  controls in the form, 17-18
  forms, 29-41
  mouse events program, 182-185
Drive list box, 31, 223-224
Drive property, 223-224

E

Edit menu, 5
editing, text editor, 232-240
Enabled check box, 92
Enabled property, 69
ENDOC method, 165
EOF function, 242
ErrorMessageLbl_Click, 58
errors
  abandon, 218-219
  defensive programming techniques, 215-216
  delayed-error-handling, 221
  fatal, 218
  handling, 216-218
  ignore, 219
  limited-retry, 220
  retry, 219-220
  retry-with-new-data, 220-221
  run time, 215-221
event-driven programming, 6-8
event-handling procedures, 24-25
  window, 19-21
  writing, 54-56
ExecuteBtn_Click, 157
Execute_Click, 58
execution files, creating, 64-65
ExitBtn_Click, 300
ExpandTopicsArray, 288

F

family procedures, 229
File list box, 31, 225-226
File menu, 4
file system controls, working with, 223-240
file viewer
  command-driven, 70-78
  control array menus for, 105
  menu-driven, 96-97
FileFib_PatternChange, 229
FileFlb_DblClick, 235
FilenameBox_LostFocus, 239
files
  binary I/O, 266, 273-275
  creating .EXE, 64-65
  input/output, 241-281
  random-access I/O, 258-259
  selecting, 225-226
  sorting utility, 245-257, 275-281
Find menu option, 97
FindBtn_Click, 75
FindCom_Click, 200-201
FindFileBtn_Click, 75-76, 231
FindFirstBtn_Click, 265
FindNextBtn_Click, 265
FindSelTextCom_Click, 201
FOR-NEXT loop, 319-320
Form files, 16-17
FORMAT$, 165
forms
  arguments, 331
  Calendar, 168-176
  documenting, 38-40
  grouping option buttons, 132-133
  input, 125-162
  methods/statements that manage, 112-113
  modal, 111
  output to, 163-164
  selecting startup, 113-114
  setting properties, 113
  variables, 334
  working with multiple, 111-123
Form_DragDrop, 194
Form_Load, 144, 148, 157-158, 185, 189, 194, 201, 209, 227-228, 232, 235, 246-247, 260, 297-298, 300-301
Form_MouseDown, 185, 189
Form_MouseMove, 185
Form_MouseUp, 185
Frame, 3
Frame control, 31
FREEFILE function, 241
function Max, 249
FUNCTION procedures, 328-329

G

GET statement, 258, 274-275
GetFileExt function, 247
GetFilename function, 247
GetFunctionName function, 247-249
GetSubName function, 249
GetTempFilename function, 249
GETTEXT method, 198-199
global variables, 334
GotoBtn_Click, 265
GotoXY, 82

H

help
  compiler, 283-292
  on-line, 283-303
Help command button, 92
Help menu, 5, 12-13
  Contents option, 12-13
  context-sensitive, 14
  Help option, 14
  Index option, 13
HelpCom_Click, 297
HelpRec user-type, 284
Hidden property, 225
HIDE method, 112
Horizontal scroll bar, 32
Hypertext, 11-12

I

I/O
  binary, 266, 273-275
  common file, 241-243
  contacts database application, 259-273
  file, 241-281
  listing sort utility, 245-257
  random-access, 258-273
  sequential file, 243-245
  text sorting utility, 275-281
identifiers, 306-308

IF-THEN structure, 315
IF-THEN-ELSE structure, 315-316
Index text box, 92
input, 125-162
INPUT statement, 245
INPUT$ function, 245
input/output (*see* I/O)
INPUTBOX$, 150-153

K

keyboard, developing low-level handlers, 211-214
KeyDown, 211-214
KeyPress, 125-132
KeyUp, 211-214
KILL statement, 242

L

Label, 3, 31
LargeChange property, 146
LINE INPUT# statement, 245
List boxes, 4, 31, 140-144
List property, 141, 224, 225
ListCount property, 141-142, 224, 225
ListIndex property, 141, 224, 225
listing sort utility, 245-257
literal constants, 305-306
LOAD statement, 112
LoadCom_Click, 233, 235
LoadCurRec, 260
LoadScreen, 82
LoadTopic, 301
LOF function, 242
loop control structures, 319-325

M

MakeBtn_Click, 171
Max property, 145
MAX_TOPICS constant, 284
measurements, converting, 114-122
Menu Design Window, 91-93
Menu Structure text box, 92
menus, 4-6
  application, 91-110
  assigning shortcut and access keys to, 98
  building, 93-95
  building command buttons, 92-93
  component definitions, 91
  control arrays, 102-105
  controlling at run time, 95-96
  file viewer, 96-97
methods, 163-164 (*see also* specific names of)
  managing forms, 112
Min property, 145
module files, 17
modules, 333-335
  selecting startup, 113-114
  variables, 334
MonthBox_KeyPress, 171
MonthName, 174
mouse events
  free-drawing program, 182-185
  managing, 181-196
  pointer program, 185-190
MouseDown event, 181
MouseMove event, 182
MouseUp event, 181-182
MSGBOX, 150-153

N

NAME statement, 242-243
NewLine, 82
NEWPAGE method, 165
Next command button, 92
NextBin_Click, 301, 303
NextPct_Click, 265
Normal property, 225
NormalCom_Click, 183-184
NowBtn_Click, 144
number systems, converting, 114-123

O

object oriented programming (OOP), 1-3
objects, properties, 8-9
OKBtn_Click, 235
ON ERROR statement, 216-218
OPEN statement, 244, 258, 273-274
Operand1Box_KeyPress, 128-129, 158
Operand1Lbl_Click, 60
Operand2Box_KeyPress, 128-129, 158
Operand2Lbl_Click, 60
OperatorBox_KeyPress, 129, 132
operators, 306
Option buttons, 31, 132-135
  grouping in forms, 132-133
  selecting, 133
Options menu, 5
output, 163-180
  Calendar form, 168-176
  pseudoscreens, 165-168
  text, 176-180

P

PasteCom_Click, 204
Path property, 224-225
Pattern property, 225
PatternBox_LostFocus, 229
Picture boxes, 30
  output to, 163-164
  pseudoscreen in, 165-168
PPrint, 82
PreviousBtn_Click, 301, 303
PrevPct_Click, 265
PRINT method, 163
PRINT statement, 164
PRINT# statement, 244
PrintBtn_Click, 178
PrintCom_Click, 185
printers, output to, 163-165
PrintHeading, 178
printing, text, 176-180 (*see also* output)
PrintTarget property, 165
procedures, exiting, 329-330
projects
  opening, 22
  saving, 21-22, 38-40
  window, 16-17
properties, 68-69, 95-96, 140-141, 145-146, 164
  altering control, 18-19
  objects, 8-9
  setting form, 113
  viewing, 43-46
  working with, 43-46
pseudoscreens, 80-86, 165-168
PUT statement, 257-258, 275

Q

QuickFilenameCom_Click, 235
QuickFilename_Click, 110
QuitBtn_Click, 27, 64, 74, 144, 171, 266

R

Radio buttons, 3-4, 132-135
ReadBtn_Click, 74-75
ReadCurRec, 260
ReadOnly property, 225
ReadOnlyCom_Click, 204
RemoveItem, 142
ResetBtn_Click, 209
RESUME statement, 216-218
ReverseBtn_Click, 26-27
Run menu, 5
run time errors, 215-221
  types of, 218-221

S

SaveCom_Click, 201, 204
SaveCurRec, 260
SaveScreen, 82
SaveThisRec, 263-264

saving
forms and projects, 38-40
projects and program files, 21-22
screens
emulating scrollable DOS, 80-86
PSCREEN project, 86-90
pseudoscreen in a picture box, 165-168
Scroll bars, 4, 68
vertical/horizontal, 144-149
ScrollUp, 82
Search menu, 5
SEEK function, 259, 275
SEEK statement, 259, 275
SELECT-CASE, 6-8, 317-319
SelStart property, 68
SelText property, 68
Separator check box, 92
SETTEXT method, 197-198
ShiftedCom_Click, 185
Shortcut Key combo box, 92
SHOW method, 112
ShowOSBtn_Click, 76
SIZE_INCREMENT constant, 284
SmallChange property, 146
SortBtn_Click, 265
Sorted property, 141
sorting utility
listing, 245-257
text, 275-281
SortKeys, 288
SortProcedureData, 249-250
SortVBfileBtn_Click, 250-251
source code
BASEFORM.FRM, 123
CAL1FRM.FRM, 176
CALEND1.FRM, 174-176
CATALOG.FRM, 252-257
CHECKBOX.FRM, 139
COMBOBOX.FRM, 145
FILEDLG.FRM, 240
FVIEW1, 76-78
FVIEW2A, 99-101
FVIEW2C, 106-109
HELPCOPM.FRM, 288-292
HELPDEMO.FRM, 298-300
HELPFORM.FRM, 302-303
INPUTBOX.FRM, 154
INTCALC.FRM, 130-132
LENGTHFO.FRM, 122
MOUSEV1.FRM, 186-187
MOUSEV2.FRM, 191
MOUSEV3.FRM, 195-196
OPTION.FRM, 136-137
PRINTFL1.FRM, 179-180
PSCREEN, 83-86
PSCREEN2.FRM, 169-170
SCICALC.BI, 61-64
SCICALC2.FRM, 158-162
SCROLLBA.FRM, 149
SORTFILE.FRM, 277-281
TEMPFORM.FRM, 122
TIMER1.FRM, 212-213
TREEVU1.FRM, 230-231
VBCARD.FRM, 267-273
WINDEDIT2.FRM, 214
WINEDIT1.FRM, 204-207
WINEDIT3.FRM, 236-239
StartBtn_Click, 209, 211
statements, managing forms, 112
static local variables, 330-331
StoreBtn_Click, 60, 154, 157
Style property, 140
SUB procedures, 325-328
symbolic constants, 308
System property, 225

T

TAB key, manipulating order, 79-80
temperatures, converting, 114-122
Text boxes, 3, 31, 67-69, 92
file viewer, 70-78
monitoring input, 125-132
text editor, 232-240
Text property, 141
text, sort utility, 275-281
TextBox_Change, 195
TextBox_DragOver, 194-195
TextBox_KeyPress, 178, 195
TextHeight method, 164
TextWidth method, 164
Timer, 4, 32
timer controls, 207-211
TimerBtn_Click, 24, 25, 148
TimerClk_Timer, 211
TimeSci_Change, 148
Toggle OS menu option, 97
Toggle Read menu option, 97
ToggleBtn_Click, 25-27, 76
ToolBox, 30-32
Tools menu, 6
TopicsLst_DblClick, 301
TOPIC_SIZE constant, 284
TrashboxPct_DragDrop, 194
TrashboxPct_MouseDown, 195
TreeDir_Change, 229, 239

U

UNLOAD statement, 112
UpdateScreenText, 82
UseFileDialogBox, 232

V

values
changing current, 47
documenting custom, 48-51
viewing, 43-46
VariableBox_KeyPress, 158
variables, 308-309
array, 309-312
form, 334
global, 334
module, 334
procedure (local), 334-335
scope of, 333-334
static local, 330-331
Vertical scroll bar, 31
View menu, 5
ViewCom_Click, 204
Visible check box, 92
Visible property, 69
Visual Basic
features, 3-4
opening/closing, 9
overview, 1

W

WhereX, 82
WhereY, 82
Window menu, 5
WRITE# statement, 244
WriteCurRec, 260

Y

YearBox_KeyPress, 171

Order Form for Readers Requiring a Single 5.25″ Disk

This Windcrest/McGraw-Hill software product is also available on a 5.25″/360K disk. If you need the software in 5.25″ format, simply follow these instructions:

- Complete the order form below. Be sure to include the exact title of the Windcrest/McGraw-Hill book for which you are requesting a replacement disk.
- Make check or money order made payable to *Glossbrenner's Choice*. The cost is **$5.00** (**$8.00** for shipments outside the U.S.) to cover media, postage, and handling. Pennsylvania residents, please add 6% sales tax.
- Foreign orders: please send an international money order or a check drawn on a bank with a U.S. clearing branch. We cannot accept foreign checks.
- Mail order form and payment to:

  Glossbrenner's Choice
  Attn: Windcrest/McGraw-Hill Disk Replacement
  699 River Road
  Yardley, PA 19067-1965

Your disk will be shipped via First Class Mail. Please allow one to two weeks for delivery.

Windcrest/McGraw-Hill Disk Replacement

Please send me a replacement disk in 5.25″/360K format for the following Windcrest/McGraw-Hill book:

Book Title _____

Name _____

Address _____

City/State/ZIP _____

CUT TO OPEN

Visual Basic for DOS

Namir C. Shammas

Book #4366

©1993 by Windcrest Books,
an imprint of TAB Books,
a division of McGraw-Hill, Inc.
Blue Ridge Summit, PA 17294

Power DOS programming made easy!

Packed with dozens of sample programs, hundreds of clear screen images, and helpful icons highlighting special tips and warnings, ***Visual Basic™ for DOS*** provides all the hands-on guidance you need to develop professional-quality DOS applications quickly and easily. After introducing you to the basic concepts behind object-oriented programming, Namir Shammas walks you through each step involved in writing a Visual Basic application, including how to:

- Input and view text with the text box object
- Create Visual Basic menus
- Create multiple forms and dialog boxes
- Use a visual form to emulate a screen
- Manipulate mouse controls more effectively
- Use the clipboard object and timer controls
- Work with sequential, random-access, and binary files
- Offer context-sensitive on-line help in your Visual Basic applications

In addition to a companion disk containing all the program code covered, ***Visual Basic™ for DOS*** includes Shammas' unique and proven system for effectively documenting Visual Basic applications.

About the Author

A professional programmer, Namir Shammas has written numerous articles for such magazines as *Computer Language, Dr. Dobb's Journal,* and *Byte.* He is the author of Windcrest's *Visual Basic: Easy Windows Programming, Visual Basic Power Programming,* and *Windows™ Batch File Programming.*

PROGRAMMING
Beginner/Intermediate

$29.95

0193

ISBN 0-8306-4375-3

90000

9 780830 643752

Windcrest®/McGraw-Hill
Blue Ridge Summit, PA 17294-0850